CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

To support this research work, many research papers have been referred and the core concepts of those papers have been observed. Based on the observations, they have been used to assess the research area and its challenges, techniques available in the related fields and their improvement areas. Based on that, this section is segregated into four sections such as software testing, test case generation techniques, test suite minimization techniques and test case prioritization techniques.

2.2 SOFTWARE TESTING

Software testing is a vital part of the software development life cycle. It span over all the development phases. One of the main challenges in software testing is maintaining and deploying a real world test platform for the projects. As a rule, following experts are included in testing of a system to an extent of their individual limits: Project Lead/Manager, Software Developer, Software Tester, and End User (Korayincki et al, 2012). Bounded-exhaustive testing is an automated approach that verifies the code under test within given bounds for all inputs. It contains three activities. First, the user defines a set of test inputs and gives test oracles to check test outputs. Second, all the inputs are generated by the tool, perform test execution on the code under test, and verifies the outputs by the oracles. Third, the users assess the failing tests to submit defect reports or investigate the code (Jagannath V et al, 2009). Testing techniques are based on pre-code artifacts, such as architecture specifications, requirements and design.
The overall testing process can be improved with the help of techniques that use these pre-code specifications for tasks such as test-case planning and development (Harold M.J, 2000).

Fuzz testing is also called as fuzzing, negative testing or robustness testing. This technique supplies random inputs to the application. (Jovanovic et al, 2008). The intend of compiler testing is to verify that the compiler implementation matches to its specifications, which is to create an object code that truly related to the semantic language and syntax as mentioned in the language documentation (Boujarwah A.S et al, 1997). Regression testing is clearly defined in the words as "Retesting a specific framework or part to validate that the changes have not brought any unwanted impacts and also the framework or test suite still follows its expected prerequisites. The author illustrated few procedures for using test execution data to experiment prioritization for regression or repetitive testing, which includes systems that need test cases focused around their overall scope of code segments (Rothermel. G et al, 2001).

Control flow graphs based test selection algorithm for regression testing, build control flow graphs for a tool or technique or system and its modified form, and utilize these graphical representations to select tests that perform the execution on changed code from the initial test suite (Rothermel. G et al, 1997). Regression testing is a part of maintenance activity of the software that is executed on the updated software to guarantee that the enhancement does not unfavourably degrade the unaltered contribution of the software. Since regression testing is performed often in software maintenance, it represents a widespread portion of the maintenance costs (Leung H.K.N, 1989). A system is needed to select an illustrative set of test cases from an unreduced test suite that should give the same scope as the original complete test suite. This choice is done by distinguishing and taking out the redundant, out of date tests in the test suite.
This strategy is outlined utilizing data stream testing strategy (Harrold M.J et al, 1993).

For instance some observations shows in particular cases that there is few or no trouble in the capability of a minimized test suite to discover defects in comparison to its un-reduced original. Other experimental observations denotes the flaw identification ability of test suites might be seriously impacted by minimization (Sampath. S, 2008). In the test approach, the software test society outlines numerous mechanisms for the test process based on all attributes that impact the effort of testing, such as available time, tool resources and skill of testers. It is recognized that the amount spend in the testing structure and processes brings to losses for many billion dollars (Tassey. G, 2002). The observations also indicate that recuperating the testing structure is not economical or straightforward to implement. Studies clarifies that the most common obstruction on acceptance of new software testing tools were measured to be the investments related with the adoption process, the time and effort the adoption process takes and the complexity of adopting new tools.

Similarly, in the implementation of testing methods, deficiency of expertise was measured as the most vital reason to put off the acceptance of new testing techniques, and training on new methods was considered as very expensive and time-taking process to invest in a reputed software developing organization. With the traditional software development models like waterfall model, the testing effort generally follows the core development stage of the software (Pfleeger, S.L et al, 2006). In this method, the testing does not expect any deviations to the strategy or expectations, but in real, the software may still need to endure changes, particularly if the client or end user has authority on the enhancement (Highsmith J et al, 2001). To address this problem, a new trend in the software development approach called agile development has been introduced (Abrahamsson P et al, 2002).
The agile methods are designated as an experiment to answer to the business society, requesting for lighter-weight and flexible software development process. The agile models varies from established, plan-driven, development models by supporting contact between stakeholders and invention of releases instead of surplus design and documentation before execution (Fowler M et al, 2001). Software testing concentrates to advance the excellence of a software product, and actually is a key factor on determining whether the software project is cost-effective (Huang L et al, 2006). However, in the dimension of excellence the definition of quality can be upsetting, as the perception of quality is closely associated to a number of individual interpretations. The meaning of quality and extensive definition work for explaining what is actual quality and how it influence product ideas such as market situations and cost-effectiveness. Garvin outlines five diverse classifications for quality; product based transcendent, manufacturing-based and user-based and value-based definition. Though they describe the similar phenomena, product quality greatly vary. For instance magnificent quality is an essential excellence, which is complete and uncompromising benchmark for high accomplishments, certainly recognized if present. On the other hand, user-based excellence is the more common user desired characterization, though manufacturing-based description establish conformance to the product expectation specifications. Garvin also debates the various definitions by indicating that various people seem to have diverse outlooks as to what comprises quality; they tend to implement the characterization they are most aware of (Garvin D.A, 1989).

The various views of quality also means that the metrics of software quality has few considerations. A Journal paper by (Jorgensen M, 1999) presents three assumptions for promoting dimension for software excellence: There are no common quality metrics, but significant metrics for particular situation. Secondly, broadly accepted quality metrics need maturity in research and thirdly,
quality metrics foresee or indirectly compute quality. In short, Jorgensen establishes that there are no universal metrics, but the methods using quality attributes, features can be used to estimate or foresee software quality. In addition to the software quality behaviours, criticality is another measure for software quality requirements. (Boehm B et al, 2003). Software criticality is an estimated indicator, indicating the likely poor outcome of software failure. Unlike other comparable metrics such as safety and integrity level or safety performance level which both computes the chance of failure against operated hours, criticality is much simple metric as it does not need exact measurements. It simply indicates the possible worst-case situation caused by the software failure.

2.3 TEST CASE GENERATION METHODS

There is a mixed approach presented, by which the known issues are determined in successful way. First select the test cases that can test some part of progressions, and afterward do the reduction. The approach has used genetic algorithm for generalized unit testing to assess the best fitting test cases (Kumar A et al, 2010). A newer version of bee colony algorithm has been used to produce pair wise test sets. This approach is based on the behaviour of biological bees. The honey bees foraging way has been taken as base to the job scheduling mechanism (McCaffrey J.D, 2009). A strategy using Genetic algorithm is used for automatic test data generation. And also the strategy by using GA to automate fault and branch based testing and offered an insight on automatic structural testing using Genetic Algorithm. A Search based technique is suggested to reduce the input domain and also introduced a broad view and approach to create dynamic test data (Sthamer. H.H, 1996).

Another modelling architecture has described for the purposes of model based testing and verification and testing. Their structural design is made up of
two components. System model was the first component of the architecture that has written in UML; this was a collection of state, class and object diagrams: the class diagram found the entities, the state diagrams gave explanation on how these entities might developed in the system, the object diagram spelled out an initial construction. Test directive was the second component which was also based on UML code. This test directive contains particular object and state diagrams, the object diagrams were utilized to declare test constraints and coverage criteria; the state diagrams have been utilized to found test purposes (Cavarra A, 2000).

The system model and test directives are able to build using any of the model tool sets like Rational Rose. This approach developed a relation framework based on choice for supporting the generation of category-partition test case. All the conditions among choices must be manually defined. They assess the conditions among choices in thorough and organized manner via various relations introduction. The approach included the following qualities like reliability checks of mentioned conditions among choices, wherever it is possible to automatically deduct the new conditions among choices, and a more effectual test suite building process. A specification-based testing method, Category Partition Method (CPM) that helps the software testers to form test cases through the functional specification refining of a program into test conditions. They make it as possible for the software tester to specify the comparative priorities for choices that were used to form the subsequent completion of test frames.

These test suites can be used subsequently as the basis for test cases generation. Their approach is applied to real-life situations and described on their effectiveness of consistency checks and automatically deduct the choice relations (Chen. T.Y et al, 2003). From formalized requirements, test cases derivations by
an automated chain in the view of object-oriented embedded software. But there was a huge space between concrete test cases and high level use cases to automate the test generation process. The test cases are created into two steps. Use cases orderings are inferred from contracts of use case, and that use case scenarios are replaced to generate test cases for each use case. Their motive was to generate test cases for vigorously detecting faults in embedded software and to cover the system in the statement coverage perspective with those generated tests (Benoit Baudry et al, 2006). Figure 2.1 represents the Test case life cycle as below.

A method of creating test cases from UML scheme diagrams has implemented, having all vital information stored for test generation in the STG, and navigate through the STG to generate test cases. The test suite creation algorithm traverses through the STG at two different levels. The search starts
with the UDG. This search stopovers all generated use cases for identifying preliminary faults. At this first level, if initialization of use case fault occurs then it is assumed that the defects are in its process and it is not necessary to execute test cases parallel to the operation. At the second level search, initializing from a use test case point the equivalent SDG was searched and test cases were created to find operational issues or defects (Sarma Monalisa et al, 2007). Auto test is a test case generator, created to initialize and run unsystematic tests for modules from a general repository. The random creation of test inputs were attractive because it was relatively suitable and economical, both in conditions of implementation time and execution effort. It is tested on all its methods, when Auto Test should start to tests a class. The results were presented through an empirical study that was based on testing Eiffel classes randomly each with origin of random number generator. This approach analysed more than 6 million failures generated during the sample tests; the proof of this study provides that random testing was predictable respective to the corresponding number of defects identified over time (Alexander pretschner et al, 2008).

The time, total expense and work effort required for global software testing rest on the whole number of test cases or size of the test suite. A test case is a set of contributions provided to the software application to get the pre-defined output. The effort usually subjective to the size of the software application and the number of test cases. There are few techniques of test case generation that relies on the applications like web application based test case generation, object oriented application, structured systems, UML applications, applications based on computational and genetic algorithms and others. Software is tested using defined set of contributions and its achievement is based on the predictable outcomes resulted from the test case criteria. All these years, several numerous diverse testing have been projected for creating test cases. Test use cases can be obtained from use cases and also be imitated from system specifications.
The novel technique for presented for test case creation based on UML classification graphical representations recycled the edge marking and dynamic slicing technique which is implemented on MDG (Message Dependence Graph) for generating the slices (Philip Samuel et al, 2008). These slices aid in estimation of order graphical representation by which the test data is produced. Some test use case representations are also used for test case generation. The behavioural and communication graphical diagram based test case creation method uses various use case diagrams. The activity graphical representation are used to create the test case conditions. Use case Diagram Graph (UDG) is a graphical representation for obtaining test cases that illustrations all the available or possible use cases in the SUT. A novel method proposed for creating the test cases for object oriented programs from UML association and activity diagrams. To find the sub-optimal test cases, genetic algorithms is used that meets the test case competence criteria. The competence criteria is used to illustrate the capability of the test use case associated to all standards like branch coverage, statement coverage, path coverage, fault coverage etc. in terms of discovering the faults in the program. Critical path method based proposed a framework for generating test cases aids the software tester in generating of test cases by purifying functional selections. It is essential to get to know about the determination of the test along with the hardware and software requirements specifications and the definitions on how to accomplish the task. The triumph criteria relies on the predictable and the actual outcomes for the test. A test case generator “Auto Test” is used to generate and perform random tests (Ratna Raju. P.D et al, 2011). In code based test generation method test cases can be generated directly from the code. This technique is mainly used while updating existing tests based on test competence criteria. This method supports to reduce the size of regression testing test suite, or prioritize test cases. A fully automatic reasonable test cases generation scheme is given using testing technique that satisfies the exchange path coverage
criteria (Shanthi.A.V.K, 2011). Figure 2.2 explains an UML based test case generation technique graphical representation.

![Figure 2.2 Test Case Generation Techniques based on UML notation](image)

A survey of test case generation methods presented that is primarily useful for GUI testing. The first method of generating test cases for GUI based applications using full interaction sequence. Second the finite state testing and analysis of graphical user interface based applications. The test case definition of graphical user interface testing and automation uses a model driven approach. The next main motive is to develop cost effective model based technique for GUI testing. A model based approach is suggested for testing GUI using hierarchical estimate transition sets. Generating test cases from GUI model is also suggested. The method suggested is using GUI run time state as feedback to generate the test cases. All the above mentioned test case generation methods for basically GUI applications and gives a clear picture about the functionality of each technique and also provides the comparison table in which each technique has been compared using some computation parameters (QuershiT.Imran Ali et al, 2013).
The dynamic path testing technique generates test cases by running the program with various possible test case values. Few authors proposed approaches for test data prioritization based on path testing. The test cases are prioritized based on the shorter path by implementing random testing method. Few focused on generating test cases by using much structural test coverage or testing criteria using evolutionary methods like Genetic Algorithm Which generally uses the Meta heuristic search based computational approach for test case generation they also propose test case generation for C language by using random testing methods. This method is needed at the time of coding a program. The UML diagrams are very helpful in testing these types of software (Korel.B, et al, 2005). Test cases can also be created by passing through from parent root to child node as breadth first OR depth first in a tree or graph structure. When all the nodes in a path from parent to child node are reached once, then that is considered as one test case. All nodes should be covered to make sure all flows in an application are covered. One flow is considered as one test case.

The recent research is being done on test case generation, minimization and prioritization using Genetic Algorithm. A GA is a test case optimization technique which can be applied to different problems. It establishes survival of the fittest method, where the best solution continues to exist. The two main requirements of GA are (a) an encoding used to show a solution of the solution field, (b) An objective fitness function which measures the goodness of a solution. This includes two testing techniques such as mutation and crossover testing. Mutation denotes mutants which are updated function to be applied on any logic or any test case to generate new test case. Crossover is also used to generate test cases which include crossing over of two different test cases to generate a new test case (Bharti Suri et al, 2003).
2.4 TEST SUITE MINIMIZATION

A divide-and-conquer approach towards test suite reduction is proposed, which concentrates on dividing approaches that are complete with respect to the optimal and minimal representative sets, from the view of critical test cases. Divide-and-conquer approach basically divide the original problem into smaller modules or sub problems, identifies optimal solutions for the modules or sub problems, and build a solution for the original problem from solution of the modules or sub problems. They imitated redundant subset and essential subset corresponding to redundant and essential test cases respectively. An essential subset consists of essential test cases and a redundant subset is that whose attained requirements can be fulfilled by other test cases. The essential subset can be included and redundant subset can be discarded to form the representative set. (Chen T.Y et al, 2003). An inspired greedy algorithm for test suite minimization is based on Formal Concept Analysis of the relation or mapping between user requirements and test cases. Concept analysis can be used for substances with isolated behaviours. For test suite reduction, test cases are assumed as objects and requirements are considered as their attributes. The coverage information of test case corresponds to the relationship between attributes and objects. The concept analysis framework analyses the context. Concept analysis finds maximum combination of objects and features which is called as contexts. Object minimization rules and attribute minimization rules are used for dropping objects and attributes (Sriraman Tallam et al, 2005).

A test suite reduction method is proposed with discriminating redundancy to reduce the loss of fault detection efficiency. It is observed that eliminating some particular redundant test cases, results in fault detection capability. It is a significant loss. So the test cases should be included which results loss in fault detection effectiveness in the reduced test set. For this, a primary and secondary coverage criterion was used. First the test cases were selected base on which
satisfies the primary requirements and then those which satisfies the secondary requirements (Dennis Jeffrey et al, 2005). A method called Test Filter is introduced in which the statement-coverage criterion is used for reducing test cases. Test cases were assigned with corresponding weights. Number of occurrences of a test case that covers different statements is referred as its weight. The non-redundant test cases were selected based on their weights. The weight has been calculated for set of test cases and primarily the higher weighted test cases have been selected. Then low weight test cases are also selected until all the requirements are mapped (Saif-ur-Rehman Khan, 2006).

Another approach introduces the call trees, for prioritizing and minimization. The program behaviour is constructed based on tree model. The reduction component finds the subset among test cases, using dynamic call tree that covers same paths of call tree. And prioritization is done to rearrange the test cases so that all the requirements are satisfied (Adam Smith, 2011). For test suite reduction, a degraded ILP Approach is introduced which developed the method to bridge the gap between the traditional heuristic approach and ILP approach. This method produced a lower bound for minimum test suite and possible solution around lower bound was searched. The representative set is considered as best result, only if the size of that set equals to the lower bound. The representative set can be considered as good result, if the size of representative set is near to lower bound. If representative set size is very far from lower bound then Integer Linear Programming needs to be used or any other classy methods to develop representative set (Zhenyu Chen et al, 2008). A new method has been introduced which is a combination of ABC integrated with Fuzzy C-Means (FCM) and PSO to generate optimal test-cases. The ABC algorithm has three different stages such as scout bee stage, onlooker bee, an employed bee stage. Initially the ABC worker bees identify the source with greatest coverage. To identify the highest usage test case, the PSO algorithm is used. The hybrid optimization based on ABC and PSO algorithms, achieves the set of test suits
using repetitive reduction of the given unprocessed test set. The result of evaluation shows that the resulted test suites are with improved quality and reduced complexity towards the test suite generation (Abraham Kiran Josep et al, 2014).

The proposed two Multi-Objective Optimization algorithms such as Binary Multi-Objective Particle Swarm Optimization along with Crowding Distance and Roullete Wheel with Harmony Search, deals with the issues of search based optimization strategies. This experimental work covers both the functional and structural testing circumstances. The experimental results clearly show the better performance of the proposed method, in terms of search based optimization when comparing with the Multi-Objective Binary Harmony Search algorithm and Non-dominated Sorting Genetic Algorithm (NSGA-II) (Luciano et al, 2015). The test suite minimization problem which is taken into consideration is NP complete. This research work proposed a method for reducing the cost of regression testing. This approach minimize the cost for each regression testing cycle which gets the improved test cases in regression testing of complex software systems and also reduces the time taken for test executions. The overall software maintenance time is reduced using the proposed approach (Mohamed Alaa El-Din et al., 2014). Features of interacting with one another in the software system are made the testing process easier. This method is known as Combinatorial Test Design (CTD) which needs manual test design and define the parameters manually. Hence the proposed system is called Interaction based test suite minimization (ITSM) reduces the test cases without any manual intervention and with less amount of feature interaction. This method has both benefits and flaws. It needs less effort on modelling of test space and do not need to mention restrictions. But the limitation is specific with the expenses of newly created test cases. This work presents the trade-off between the two discussed methods above and the difficulties faced during the implementation (Dale Blue, et al., 2013).
The test suites get grow when the new test cases are appended to the existing test suite when adding the new functionality to the software. It is tedious to manage and execute the large size of test suites. Many test suite reduction methods are introduced to resolve this issue by eliminating the redundant test cases on considering single condition, and preserve the overall efficiency of the minimized test suite. But the proposed (Hwa-You Hsu, 2009) system is implemented by the MINTS tool which easily resolves the minimization problem, accepts multiple criteria during the elimination of redundancies and use the leveraging up to date, integer linear programming solver to identify the optimal solutions. In addition, this work also presents the procedure to handle the MINTS tool in various test suite minimization problem. A software testing needs large number of test cases to achieve the better software quality. But this would cause increase in test suite size which leads to the larger size of test suites. Many test suite minimization techniques are introduced and exist to resolve this issue, but they are less efficient in performing fault detection. Hence this work proposes a greedy algorithm which chooses the test cases with minimum overlap in requirement coverage and meets the maximum number of specified testing requirements. This algorithm is applied on the Space programs and Siemens suite to evaluate its performance (Saeed Parsa et al, 2010).

2.5 TEST CASE PRIORITIZATION

The Risk of any test case failure is determined by two factors such as probability and impact. Probability is the chances of specific scenario happening in production or how frequently or in what percentage of customers might hit that scenario. The percentage defines higher the probability. Impact determines the effect to customer or end user or company if the scenario is broken down. The higher the impact defines the higher the severity. Therefore risk is defined as Risk = Probability + Impact.
Figure 2.3 shows the probability and impact proposition. For any test case, whose failure falls under high probability and high impact category should be the test cases that need first and high priority. Similarly any test case, whose failure falls under low probability and low impact category which is the lower left corner are the test cases that need lower priority. Rest all test case scenarios should be considered as equal prioritization in general.

![Figure 2.3 Probability and impact proposition](image)

Glow worm Swarm Optimization (GSO) method is presented to solve the test suite prioritization problem using efficient SI method. This proposed work focuses on updating the searching field by glow worm at the stage of location updates itself. Depending upon the Specific Update domain search based GSO (SU-GSO) method, after updating the search field, Software under Test (SUT) is used to obtain the optimal test cases. The main object of this proposed work is to maximize the fault coverage and path coverage for obtaining the optimal
prioritized test cases. The experimental outcomes show the promising result in terms of producing an optimal ordering of test suite (Beena Raman et al, 2015). Another method, which has two level of prioritization, has been suggested for selecting test cases and sequencing them. In first level, the modified code blocks have been analysed and present the comparative connection or communication with the other modules. First level of prioritization is processed depending on the number of connected modules. Then the prioritized modules are assessed in terms of their criticality. The criticality is categorized based on the type of fault or the type of error in the particular module. Based on the criticality, the cost is evaluated to prioritize test cases. At the final level of this process, the dynamic programming method is implemented to find the appropriate sequence of test cases, thus the cost of regression testing is reduced. This work only applies the test case prioritization as EFSM method, with minimal contribution to minimization (Monika et al, 2013).

Another method with Extended Finite State Machine (EFSM) model is proposed for test case minimization process. In this work the dynamic dependencies such as control dependency and data dependency are analysed along with their different communication patterns. This proposed approach is named as dynamic interaction-based prioritization which adjusts the existing approach to improve the fault detection potential and it also considered about the optimization process to minimize the cost of resource. The involvement of this work is towards prioritization, whereas the minimization results in terms of most generic criteria like size and coverage of the test suite were less efficient (Chris Nitin Adonis Petrus et al, 2013).

One another method proposed for test suite minimization which has four factors to decide the weight of the requirement. These four factors are in brief as follows. First factor is Business value Measure, this is the factor in which the ranks are assigned to requirements based on their importance. Most critical
requirement is assigned with higher number and least important requirement is assigned with the lowest number. Second factor is the Project Change Volatility, this factor is based on the number of times the client is changing the project requirements during the various level of software development phases. One of the main reasons of a project failure happens to be the lack of requirements and changing the requirements very often. The third fact is the complexity of development, efforts of development, and the technology chosen for development, environmental situations and the time taken or required which conclude the complexity of the development phase. The fourth factor is eliminating faults of Requirements. This is the factor for assigning weights to requirements which considers the requirements error reduction, based on the historical data such as requirement failures reported by the client (Varun Kumar et al 2010). A new approach called JUPTA which operates in the absence of coverage information, has been proposed for prioritizing test cases which is widely used in java programs under the JUnit framework. It also analyses the static call graphs of JUnit test cases. Further, it calculates the ability of each test case to attain code coverage and arrange the test cases in a sequence based on those estimates (Zhang.L et al, 2009).

In general, the SDLC process consists of the following phases, which are: requirement gathering, design and analysis, development, testing and maintenance (Agrawal. H et al, 1993). Testing phases consist of the following processes: test planning, test development, test execution and evaluation of results (Rothermel. G et al, 2001). Additionally it is mentioned that the test case prioritization procedure is required for software testing, because the regression testing phase consumes a lot amount of time and cost to run, and there is no sufficient time or resources to run the whole test suite, therefore there is a need to conclude which test cases to run first (Dennis Jeffrey et al, 2006). Test case prioritization techniques schedule and prioritize test cases in a sequence that attempts to maximize defined objective function. For instance, software test
engineers might want to order test cases in a sequence that achieves code coverage at the possible fastest rate, attains features in the order of expected frequency of usage, or obtains subsystems in an order that imitates their historical inclination to fail. When the time needed to execute all test cases in a test suite is less, test case prioritization may not be cost effective. It may be most measureable simply to list test cases in any order. When the time needed to run all test cases in the test suite is long enough, the prioritization plays key role. Test case prioritization techniques provide an approach to list and run test cases, which have the highest priority to provide earlier fault detects. This study presents many techniques developed, between the years 2002 and 2008, which can help to improve a test suite’s rate of fault detection.

With active test case prioritization techniques researched in the period of 1998-2008, the research works introduce and sort out a new 4C classification of those available techniques, based on their characteristic of prioritization algorithms. The classifications are Customer Requirement-based techniques, Coverage-based techniques, Cost Effective-based techniques and Chronographic history-based techniques. Customer requirement-based techniques are approaches to prioritize test cases based on requirement specification documents. Many researchers have already analysed this area. Also, many weight features have been used in this technique, including requirement complexity, custom-priority and requirement volatility. Coverage-based methods are techniques to prioritize test cases based on the coverage criteria, such as statement coverage, total requirement coverage, requirement coverage and additional requirement coverage. Many researchers also have examined this area similar to the customer requirement based method. Cost effective-based approaches are methods to prioritize test cases based on their costs, such as cost of prioritization and cost of analysis. Many researchers investigated this area similar to the above two methods. Chronographic history-based approaches are techniques to prioritize
test cases based on their prior test execution history. Only a few researchers attempted this area for the research.

A cost awareness model for the test case prioritization and fault severities has been introduced, which exposed in the previous test execution. As well as it does not do any major change from one conclusion to another (Mohamed A Shameem et al, 2013). One another framework proposed a measure for calculating the rate of fault identification. This algorithm detects the faults in previous executions and the efficiency of prioritized test cases are evaluated with the non-prioritized cases using Average Percentage of Fault Detection (APFD) (Hla K.H.S et al, 2008). Another method proposed to prioritize the new test cases by calculating the requirements of risk coverage value and also analysing the risk objects. Further it estimates the related test cases and determining the priority of test case through the assessed values. Moreover, the effectiveness of the proposed technique has been demonstrated through experimental studies, in terms of fault severity and Average Percentage of Fault Detected (APFD) (Yoon M et al, 2012). A new test case prioritization technique has projected a Spectrum-based multiple fault localization method to identify the fault location. This suggested model describes prioritization criteria for GUI and web applications based on an event driven software. The whole purpose is to change the model and to extend a unified theory about testing of event driven software (Abreu. R et al, 2009).

Another model is proposed for test case prioritization that prioritizes the test cases based on six factors. Those factors are as listed as Changes in requirement, Customer priority, Complexity of implementation complexity, application flow, usability, and fault impact. This prioritization technique is assessed in three phases with industrial and student projects. The results are favourable to improve the rate of fault detection and its severity (Krishnamurthi. R et al, 2009). There is another cluster-based test case prioritization technique is proposed where the test
cases are clustered based on their behaviour of dynamic runtime. Significantly the required number of pair-wise comparisons has been reduced by researchers. Researchers proposed a value-driven method for system-level test case prioritization which prioritizes the test requirements. In this proposed method, there are four factors to prioritize the test cases: rate of requirements volatility, fault detection, implementation complexity and fault impact (Raju. S et al, 2012).

Prioritization is based on the fault severity. In this method, requirements to be tested first are assigned with the higher weight and the test cases covering those requirements are given higher execution priority. In this approach, the requirements are considered based on fault severity. The higher weight is assigned for the maximum number of times the fault can occur in the code and that code is required to be tested first. There are four factors are proposed to decide the weight of the requirements. These four factors are described in brief below:

1) Business value Measure - This is the factor in which the requirements are assigned rank according to their significance. Most critical requirement is assigned with higher number as 10 and least important requirement is assigned with the lowest number as 1.

2) Project Change Volatility- This factor depends upon the number of time consumer is updating the project requirements during the software development cycle. The major reason of the project failure happens to be by the lack of valid inputs and the updating and incomplete requirements.

3) Development Complexity – The complexity of the development phase is decided by the Development efforts such as manual resource effort, machinery efforts, and the technology used for development, ecological constraints and the time consumed.
4) Fault Proneness of Requirements - This is the very critical and important factor for assigning requirement weight. This factor considers the error prone requirements based on their historical data such as the requirement failure reported by the customer. Thus the importance of testing a requirement at top with high priority is calculated or assessed on the basis of weight assigned to each requirement. This is the first step of this proposed method. Then the severity measure to each fault is assigned in the second step. This is to calculate the total severity of the detected faults. This is sum of the measures of all faults detected. This is how the test cases are prioritized by the proposed approach. Requirements are prioritized first, then test cases for each given requirements is mapped, then the test suite is executed according to the priority assigned to the test cases in the test suite and the results have been assessed based on the severity of faults detected.

Prioritization in case of regression testing - In this section the test case prioritization in case of regression testing is discussed. Nine techniques have been presented to be used for this purpose. These approaches are examined in brief as follows:

1) No prioritization - In this case, no new techniques or methods are implemented. It is been used as an unprocessed test suit and it provides as a control.

2) Random prioritization - This method is applied to have further control in studies, where the test cases are arranged randomly in the test suite.

3) Optimal prioritization - In this approach, known faults are used. Its results can be used to quantify the effects of other prioritization methods which are to be used.
4) Total statement coverage prioritization - This method’s mechanism implement the program with any test case and find out, which statements were covered by those test cases; then these test cases can be prioritized on the basis of number of statements they could cover. If more than one test case covers equal or same number of statements, then we need to use some additional criteria or can change their order randomly.

5) Additional statement coverage prioritization - This approach covers the flaws of total statement coverage method and iteratively chooses a test case which gives the highest statement coverage, then updates the coverage information on the remaining test cases to find their statements that are not yet covered. Until at least one test case that covers all the statements is figured out, this whole process is repeated.

6) Total branch coverage prioritization - This method is similar to statement coverage prioritization, but it just uses the measured test coverage in the form of program branches instead of case of statement in other method.

7) Additional branch coverage prioritization - This mechanism is similar to additional statement coverage prioritization, but the only difference is that the measure test coverage is used for program branches and not in statements.

8) Total fault-exposing potential prioritization - Another method like statement and branch coverage exist and checks that whether a statement or branch is able to be reached by some test cases or not at all able to be reached not as some faults are more commonly seen as compared with others. As some test cases can render faults more easily than others, it is known as fault-exposing possible of a test case.

9) Additional fault-exposing potential prioritization – Similar to the additions that were made in total and branch coverage prioritization, additional
extensions were made in the total fault-exposing probable prioritization. As a result, this method is created where we broaden total FEP to reduce additional FEP prioritizations.