CHAPTER-1

INTRODUCTION

1.1 SOFTWARE TESTING

Software testing can be stated as the process of validating and verifying that a computer/program/application/product meets the requirements that guided its design and development. A purpose of testing is to detect software failures so that defects may be exposed and corrected. The scope of software testing often includes analysis of code as well as execution of that code in various environments and conditions as well as examining the aspects of code. There are many approaches to software testing, Reviews, walkthroughs, or inspections are referred to as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing.

The goal of software testing is to execute the software system, locate the faults that cause failures, and improve the quality of the software by removing the detected faults. Testing is the primary method that is widely adopted to ensure the quality of the software under development. According to the IEEE definition, a test case is a set of input data and expected output results which are designed to exercise a specific software function or test requirement. During testing, the testers, or the test harnesses, will execute the underlying software system to either to examine the associated program path or to determine the correctness of a software function. It is difficult for a single test case to satisfy all of the specified test requirements. Hence, a considerable number of test cases are usually generating and collected in a test suite.

Testing is the process of evaluating system or its components with the intent to find whether it satisfies the specified requirements or not. In simple words, testing

is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

According to ANSI/IEEE standard, Testing can be defined as – A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item. The flow diagram for software testing shown in fig1.1.

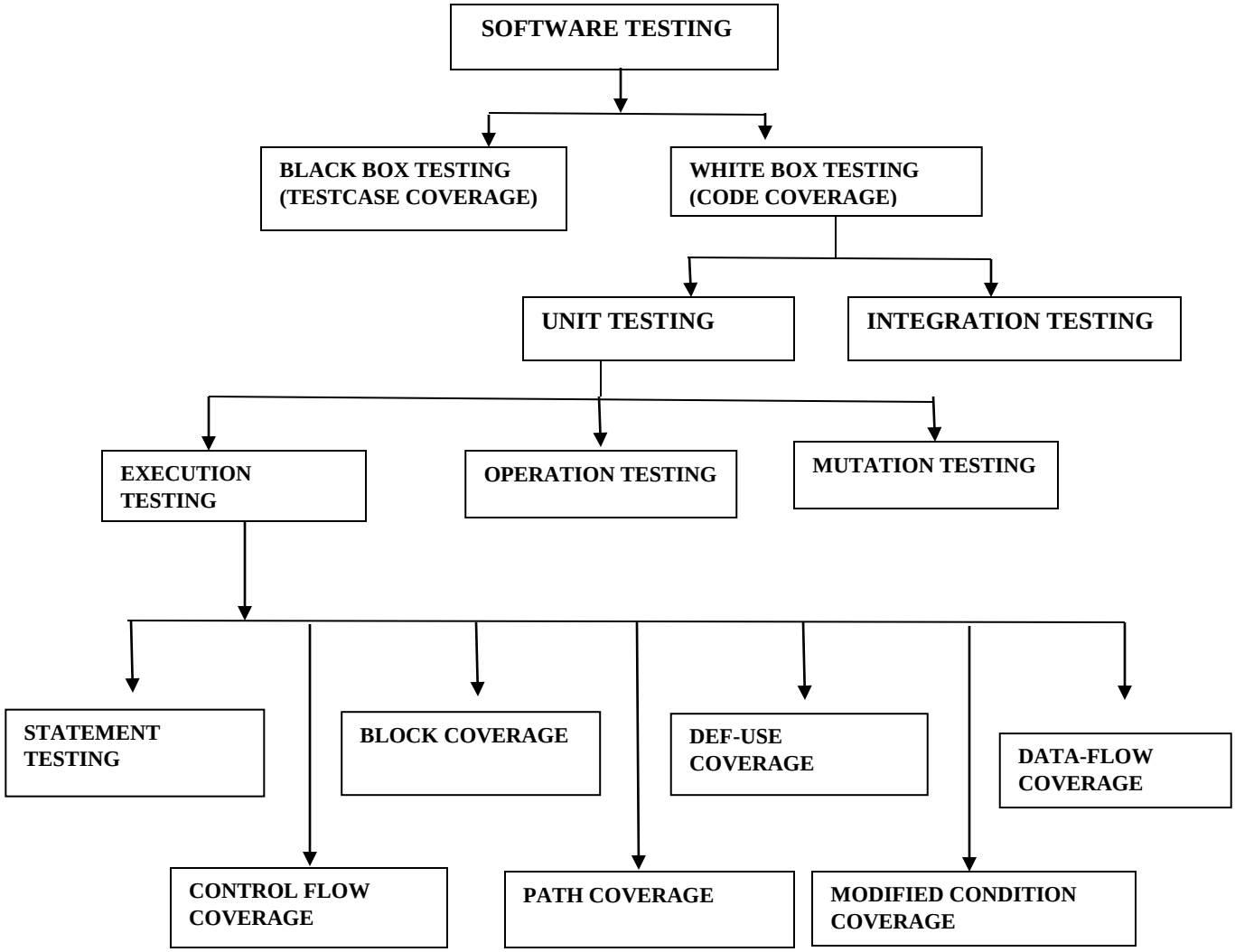

Fig 1.1 Flow diagram for software testing

1.2 WHITE BOX TESTING

Software testing have two types, there are black box testing and white box testing. White box testing technique analyzes the internal data structures, design, code structure and the working of the software. The white box testing techniques are,

1. Statement coverage
2. Block coverage
3. Def-use coverage
4. Control flow coverage
5. Modified decision coverage
6. Path coverage.

1.2.1 COVERAGE METRICS

Coverage metrics measures the number of lines of source code executed during a given test suite for a program. Tools that measure code coverage normally express this metric as a percentage. Code coverage is a measurement of how may lines/blocks/arcs of your code are executed while the automated tests are running. Code coverage metrics – coverage from unit tests. The list of four coverage is Block coverage equivalence, Control flow coverage divergence, Def-use coverage equivalence and Data flow Coverage divergence.

1.3 REGRESSION TESTING

Regression Testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the

modifications. Both during development and after development, and software is modified for several reasons, including bug fixing, functionality enhancement, and adaptation to changes in the software's operating environment, One of the most expensive activities that occur as the software is modified and enhanced is the (re)testing of the software after it has changed. This process is known as regression testing. Because regression testing is expensive, researchers have proposed techniques to reduce its cost. One approach reduces the cost of regression testing by reusing the test suite that was used to test the original version of the software. Rerunning all test cases in the test suite, but to apply a regression test selection technique to select an appropriate subset of the test suite to be run. The regression testing types shown in fig 1.2.

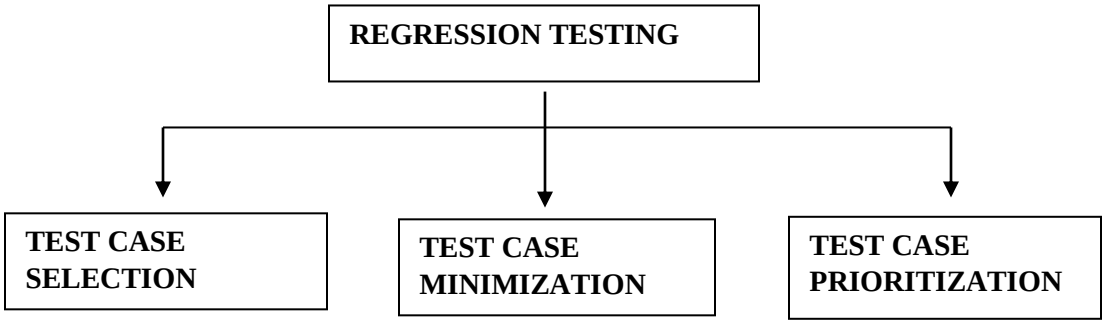

Fig 1.2 Regression testing types

Regression testing is required when there is a Change in requirements and code is modified according to the requirements, new feature is added to the software, Defect fixing, Performance issue fix. Software maintenance is an activity which includes enhancements, error corrections, optimization and deletion of existing features. These modifications may cause the system to work incorrectly. Therefore, Regression testing becomes necessary. Regression testing is the re-execution of a particular subset of tests that has been formerly performed. In

regression testing, the number of regression tests increases with the progress of integration testing, and executing each test for every program function whenever changes occur is both impractical and inefficient. Regression test suites are often simply test that software engineers have previously developed, and that have been saved so that they can be used later to perform regression testing. The regression testing process is shown in figure1.3.

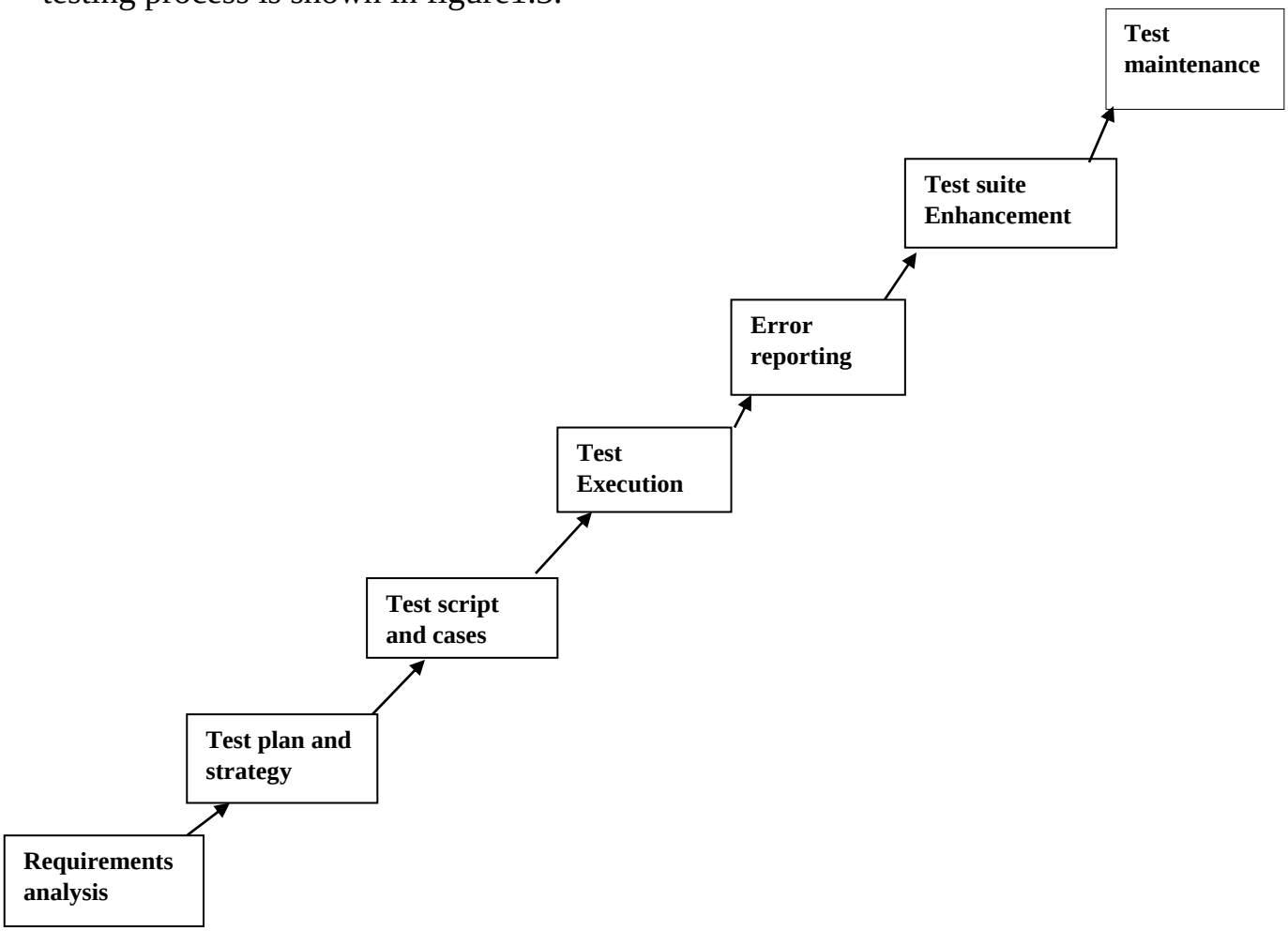

Fig 1.3 Flow Diagram of Regression Testing.

1.3.1 TEST CASES

A test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular program path

or to verify compliance with a specific requirement. Test cases underlie testing that is methodical rather than haphazard. A battery of test cases can be built to produce the desired coverage of the software being tested. Formally defined test cases allow the same tests to be run repeatedly against successive versions of the software, allowing for effective and consistent regression testing.

In order to fully test that all the requirements of an application are met, there must be at least two test cases for each requirement: one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases. Keeping track of the link between the requirement and the test is frequently done using a traceability matrix. Written test cases should include a description of the functionality to be tested, and the preparation required to ensure that the test can be conducted. A formal written test-case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post condition.

1.3.2 TEST SUITE

In software development, a test suite, less commonly known as a 'validation suite', is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors. A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

In model-based testing, one distinguishes between abstract test suites, which are collections of abstract test cases derived from a high-level model of the system

under test, and executable test suites, which are derived from abstract test suites by providing the concrete, lower-level details needed to execute this suite by a program. An abstract test suite cannot be directly used on the actual system under test (SUT) because abstract test cases remain at a high abstraction level and lack concrete details about the System Under Test (SUT) and its environment. An executable test suite works on a sufficiently detailed level to correctly communicate with the System Under Test (SUT) and a test harness is usually present to interface the executable test suite with the System Under Test (SUT).

A test suite for a primarily testing subroutine might consist of a list of numbers and their primarily (prime or composite), along with a testing subroutine. The testing subroutine would supply each number in the list to the primarily tester, and verify that the result of each test is correct.

1.3.3 TEST CASE SELECTION

One of the main types of regression testing, which is also considered its biggest field is Test Case Selection. The concept of Test Case Selection comes in tandem with the definition of Regression Testing. As elaborated earlier it is both expensive and impractical to simply retest all the test cases after software or program has updated. Thus a method is required in order to identify and select these test cases that are relevant based on a particular criterion.

1.3.4TEST CASE MINIMIZATION

Test case minimization is to generate representative set from test suite that satisfies all requirements as original test suite with minimum number of test cases. Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. The minimization problem described by Definition 1 can be considered as the minimal

hitting set problem. Note that the minimal hitting set formulation of the test suite minimization problem depends on the assumption that each can be satisfied by a single test case. In practice, this may not be true. For example, suppose that the test requirement is functional rather than structural and, therefore, requires more than one test case to be satisfied.

The minimal hitting set formulation no longer applies. In order to apply the given formulation of the problem, the functional granularity of test cases needs to be adjusted accordingly.

example: A test suite, T, a set of test requirements {r1, . . . , rn}, that must be satisfied to provide the desired 'adequate' testing of the program, and subsets of T, T1, . . . , Tn, one associated with each of the r is such that any one of the test cases tj belonging to Ti can be used to achieve requirement ri . Problem: Find a representative set, T 0, of test cases from T that satisfies all ris. The testing criterion is satisfied when every test requirement in {r1, . . . ,rn} is satisfied. A test requirement, ri, is satisfied by any test case, tj , that belongs to the Ti , a subset of T. Therefore, the representative set of test cases is the hitting set of the Tis. Furthermore, in order to maximize the effect of minimization, T 0 should be the minimal hitting set of the Tis. The minimal hitting set problem is an NP-complete problem as is the dual problem of the minimal set cover problem. While test case selection techniques also seek to reduce the size of a test suite, the majority of selection techniques are modification-aware. That is, the selection is not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program.

Test cases are selected because they are relevant to the changed parts of the System Under Test (SUT), which typically involves a white-box static analysis of the program code.

Throughout this survey, the meaning of ‘test case selection problem’ is restricted to this modification-aware problem. It is also often referred to as the Regression Test case Selection (RTS) problem. More formally, following Rothermel and Harrold, the selection problem is defined.

1.3.5 TEST CASE PRIORITIZATION

Test case prioritization is a method to prioritize and schedule test cases. The technique is developed in order to run test cases of higher priority in order to minimize time, cost and effort during software testing phase.

Test suite is a container that has a set of tests which helps testers in executing and reporting the test execution status. It can take any of the three states namely Active, In progress and completed. A Test case can be added to multiple test suites and test plans.

The term "test prioritization" refers to the subjective and difficult part of testing that allows testers to manage risks, plan tests, consider cost value, and be analytical about which test to run in the context of the specific project. This process is well known as Test Case Prioritization, which is a method of prioritizing and scheduling. This technique is used in order to run test cases of higher priority in order to minimize time, cost and efforts.

Furthermore, test case prioritization provides assistance with regression testing and improves its performance. Through this method, testers can easily run test cases, which have the highest priority and provide earlier defect faults. Also, an improved rate of fault detection during the testing phase, allows faster feedback of the system that is under test. With test case prioritization software engineers can get assistance in correcting faults earlier than might otherwise be possible.

Moreover, to decide the priority of test cases, various factors depending upon the need of the software are decided.

Test case prioritization is best technique to ensure the effectiveness as well as the quality of a product, during its development and testing process. It is one such method that prioritizes and schedules test cases according their highest and lowest requirement. Through test case prioritization, testers can effectively make sure that the most important test cases are executed first.

This way important problems and defects can be found in the software or application as early as possible. Moreover, this testing provides assistance in allows testers to order their test cases, so that test cases with highest priority are executed in the testing process, than the lower priority test cases. In short, this subjective and difficult part of testing is about risk managementand being analytical about the tests that are required to be executed in the context of the project that is specified.

1.6. ORGANIZATION OF THE CHAPTER

The chapters are organized as follows, Chapter 2 discussed about the various literatures survey paper and then several minimization and prioritization techniques. Chapter 3 discussed about code coverage metrics such as block, control flow, Def-use and data flow coverage. Chapter 4 discussed about system architecture, its description and dataflow diagrams. Chapter 5 briefly explain the proposed techniques and its description. Chapter 6 discussed about experimental and analysis. Chapter 7 discussed about conclusion on the proposed approach.

CHAPTER 2

LITERATURE SURVEY

2.1 A STATEMENT-COVERAGE BASED TEST CASE REDUCTION TECHNIQUE

The paper [1] defines software testing is an important but expensive phase of software development life cycle. During software testing and retesting, development organizations always desire to validate the software from different views. But exhaustive testing requires program execution with all possible combinations of values for program variables, which is impractical due to resource constraints. For many applications, it is possible to generate test cases automatically. But the core problem is the selection of effective test cases necessary to validate the program during the maintenance phase. This target can only be achieved by eliminating all the redundant test cases from the generated pool of test suites. A novel test case reduction technique called Test Filter that uses the statement-coverage criterion for reduction of test cases. It is beneficial to optimize time & cost spent on testing and it is also helpful during regression testing.

The main concept is reduce the test case. IEEE standard defined test case is a set of test inputs, execution, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. While a test suite is a combination of test cases. The quality of a test suite is measured through following four factors: 1) its fault coverage, 2) its code coverage, 3) its size, and 4) the number of faults detected by the most effective test contained in it. Test case reduction technique help to find out effective test cases during maintenance phase and drastically reduce the testing

cost. It focuses on reducing test suites to obtain a subset that yields equivalent coverage with respect to some criteria.

Test Filter is a simple technique that picks non-redundant test cases very efficiently based on their weights. Calculate the weight of a test case is the number of its occurrences in the set of test suites. Choose highest weight test case to lower for validation purposes.

A new test case reduction technique that uses weight criteria for reduction purposes. Test Filter consumes fewer resources that is storage requirements, number of CPU cycles for selection of test cases. Ultimately, Test Filter technique reduces the size of test suites by eliminating unnecessary test cases, and also decreases the test case storage, management, and execution cost. It provide test manager with efficient and cost-effective test case reduction technique. Ultimately it improves the testing effectiveness.

2.2 CODE COVERAGE TECHNIQUE TO FIND DIVERSE TEST CASES

The paper [2] defines a software deliver to the client, the client needs to add some features the software modified, new test cases are added to the test suits, the test suite grows and the cost of regression testing increases. This paper defines a technique to solve these problems and make the testing cost effective. It introduce a set of test case comparison metrics algorithms which will quantitatively calculate the diversity between any arbitrary test case pair of an existing test suite. Our procedure mainly focuses on branch coverage criteria, control flow of a program, variable definition-usage and data values. By using these information's a signature values is calculated to find how much the test cases are diverse from each other and accordingly we can make a cluster of similar test cases together, that can effectively test under time constraints.

Test suite, for testing and validating software program can contain large number of test cases to execute various part of the software program code. The main aim is to check for defect and code compliance. The general size of a test suite can vary from hundreds of test cases to more than a million for large and/or evolved software program. Thus, test execution can take a great deal of time to complete. Additionally, test cases are often developed and thus one or more test cases in a test suite may be redundant. In that they execute the same code path for the same or similar data sets. Moreover, as the software program is updated and modified, test cases in a test suite can become duplicative. Thus, it would be advantageous to identify and eliminate redundant test cases in a test suite, allowing reduction in the test suite size which, in turn, can decrease testing time and contribute to optimize maintenance effort for the test suite.

Regression testing is an important activity to the development and maintenance of evolving software. Difficulty is that, it requires large amount of test cases to test new or modified parts of the software. Test-suite reduction techniques attempt to reduce the costs of saving and reusing test cases during software maintenance by eliminating redundant test cases from test suites.

Coverage metric are used to identify the duplicate test case. The four coverage metric are block coverage, control flow coverage, de-use coverage, data flow coverage. Using four coverage value signature value is calculated. Block coverage and De-use coverage are used to calculate the similarity of two test case. Control flow and Data flow are used to calculate the diversity of the test case. Signature values represent how much the test cases are similar or distinct from each other. For this threshold values for equivalence signature and divergence signature are defined. According to these threshold values, weather two test cases are allowed to make a group or not can be easily defined. Two test cases are

deemed similar to group or cluster if the divergence signature value for the test case pair is less than or equal to a pre-defined divergence threshold and the equivalence signature value for the test case pair is greater than or equal to a pre-defined equivalence threshold. Finally similar test case are identified and omitted.

2.3 EXTRACTING TEST CASES BY USING DATA MINING

The paper [3] defines Case-Based Reasoning and Data mining are used as efficient methods for effort estimation and automated testing have been investigated respectively. Software has many outstanding features but does not work properly due to lack of testing. The test results could help the developer to classify them in different categories such as different process models and different types of errors in each developing life cycle phase, then by having these classified results and using data mining methods and Case-Based Reasoning. It would be easy to have the new software's properties and estimate the future test cases in order to reduce the cost of testing phase and eventually the developing cost in similar upcoming projects. In this paper testing the similar software with similar test cases. It is help to identify the test case reduction reduce the cost. It evaluates any upcoming software by fining the most similar case for it from the stored cases and do the performed test cases for it. By estimating the proper set of domains for each attributes, it could increase the efficiency.

Case-Based Reasoning (CBR) is able to utilize the specific knowledge of the previously experienced, concrete problem situation (cases). A new problem is solved by finding a similar past cases and reusing it in the new problem situation. Case-Based Reasoning is to solve a new problem by remembering a previous similar situations and by reusing information and knowledge of that situation.

Data mining is defined as a process used to extract usable data from a larger set of any raw data. It implies analyzing data patterns in large batches of data using one or more software. Data mining as applications in multiple fields, like science and research. Cluster analysis is the organization of a collection of patterns (usually represented as a vector of measurements, or a point in a multidimensional space) into clusters based on similarity.

Testing could be done automatically and manually, it doesn't matter which technique has been considered, the important factor is how to extract the attributes and gather the most accurate set of data which can be used as a new testing case for similar future software. Representation of case based process as shown in Fig 2.1.

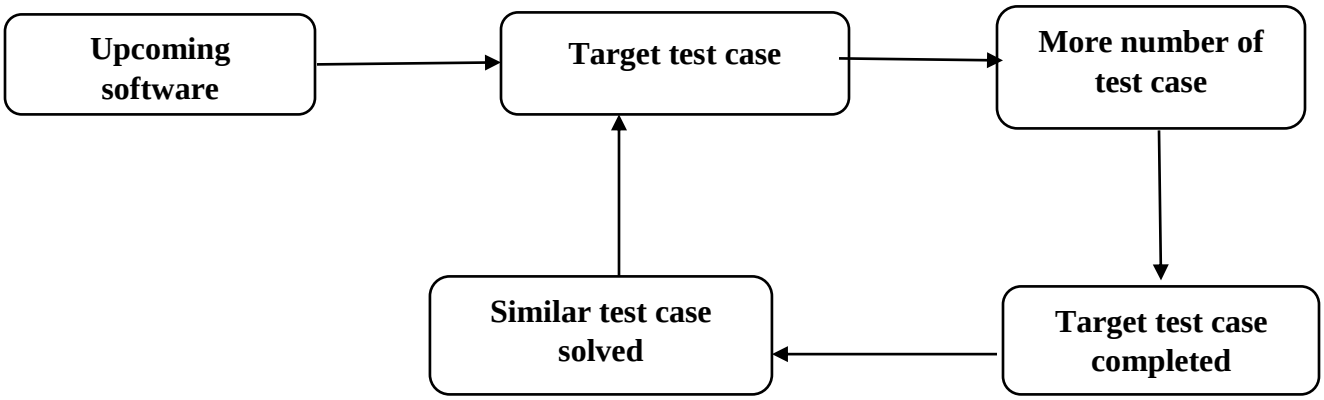

Fig 2.1.Test case based reasoning process

2.4 A DATA FLOW COVERAGE TESTING TOOL FOR C

The paper [4] defines software testing is to detect errors in a program and, in the absence of errors, gain confidence in the correctness of the program. Thorough testing requires both fictional and coverage testing. Functional testing attempts to assure that a program's specifications are met by exercising the features described in the specification. This kind of testing depends only on the specifications of the program and is independent of the encoding of the program. Coverage testing identifies constructs in the program encoding that have not been exercised during

testing. Coverage testing helps the tester create a thorough set of tests and gives a measure of test completeness. It describes a tool, ATAC (Automatic Test Analysis for C), which supports data flow coverage testing for C programs.

The use of ATAC in Bell core development increases we hope to collect data on the actual usefulness of data flow coverage testing to real code. We intend to investigate the relationship between the quality of data flow testing and the subsequent detection of field faults. One would hope, mutates mutandis, that code tested to 85% c-uses coverage would exhibit a lower field failure rate than similar code tested to 20% c-uses coverage. The establishments of a correlation between good data flow testing and a low rate of field faults (or that there is none) is the ultimate and criticaltest of the usefulness of data flow coverage testing. This method only useful for C program.

2.5 REGRESSION TESTING MINIMIZATION, SELECTION AND PRIORITIZATION

The paper [5] defines regression testing is a testing activity that is performed to provide confidence that changes do not harm the existing behavior of the software. Test suites tend to grow in size as software evolve, often making it too costly to execute entire test suites. A number of different approaches have been studied to maximize the value of the accrued test suite: minimization, selection and prioritization. Test suite minimization seeks to eliminate redundant test cases in order to reduce the number of tests to run. Test case selection seeks to identify the test cases that are relevant to some set of recent changes. Test case prioritization seeks to order test cases in such a way that early fault detection is maximized.

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. Test

case selection, or the regression test selection problem, Is essentially similar to the test suite minimization problem, both problems are about choosing a subset of test cases from the test suite. The key difference between these two approaches in the literature is whether the focus is upon the changes in the System Under Test (SUT). Test suite minimization is often based on metrics such as coverage measured from a single version of the program under test System Under Test (SUT). Test case prioritization seeks to find the ideal ordering of test cases for testing, so that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point. Goal of test case prioritization remains that of achieving a higher fault detection rate, prioritization techniques actually aim to maximize early coverage.

This logic provides both a survey and a detailed analysis of trends in regression test case selection, minimization and prioritization. The paper shows how the work on these three topics is closely related and provides a survey of the landscape of work on the development of these ideas. This paper cover basic concept of regression testing and it types only.

2.6 MECHANISM FOR IDENTIFICATION OF DUPLICATE TEST CASES

The paper [6] defines selection of distinct test cases and elimination of duplicates is two major problems in software testing life cycle. In order to solve these and make the testing cost effective, a set of test case comparison metrics algorithms is introduced which will quantitatively calculate the difference between any arbitrary test case pair of an existing test suite. Metric values for test case pair are generated and combined to create one or more unique signature values. Using these signature values a cluster of similar test cases are generated, that can effectively test under time constraints.

Test suite, for testing and validating software program can contain large number of test cases to execute various part of the software program code. The main aim is to check for defect and code compliance. The general size of a test suite can vary from hundreds of test cases to more than a million for large and/or evolved software program. Thus, test execution can take a great deal of time to complete. Additionally, test cases are often developed and thus one or more test cases in a test suite may be redundant. In that they execute the same code path for the same or similar data sets. Moreover, as the software program is updated and modified, test cases in a test suite can become duplicative. Thus, it would be advantageous to identify and eliminate redundant test cases in a test suite, allowing reduction in the test suite size which, in turn, can decrease testing time and contribute to optimize maintenance effort for the test suite.

Regression testing is an important activity to the development and maintenance of evolving software. Difficulty is that, it requires large amount of test cases to test new or modified parts of the software. Test-suite reduction techniques attempt to reduce the costs of saving and reusing test cases during software maintenance by eliminating redundant test cases from test suites.

Block coverage equivalence measures block testing overlap between two test cases of a test suite. Control flow divergence measures the similar our path selection criteria are based on an investigation of the ways in which values are associated with variables, and how these associations can affect the execution of the program. Two test cases that test the same blocks that have conditional path within them. DU equivalence measures Def-use (definition or use) path testing overlap between two test cases in a test suit. Data divergence is computed as the similarity of the number of times two test cases execute the same conditional branches, loops and or blocks. Counts are used to indirectly represent the data

values of variables in the software code under test. Each test case signature is an aggregate quantifiable metric that is used to identify the amount of similarity or dissimilarity of the test cases of a test case pair.

This method we address the four test case comparison metrics algorithms and its implementation on small isolated code. By using these algorithms difference level in terms of signature values between two test cases are calculated. These papers totally cover the four coverage metrics and also calculate the signature value for each test cases. This value only useful for identify the duplicate test cases. Here, test case minimization testing is need to effective regression testing.

2.7 A SURVEY ON TEST SUITE REDUCTION FRAMEWORK AND TOOLS

The paper [8] defines Software testing is a widely accepted practice that ensures the quality of a System under Test (SUT). However, the gradual increase of the test suite size demands high portion of testing budget and time. Test Suite Reduction (TSR) is considered a potential approach to deal with the test suite size problem. Moreover, a complete automation support is highly recommended for software testing to adequately meet the challenges of a resource constrained testing environment.

Several TSR frameworks and tools have been proposed to efficiently address the test-suite size problem. The main objective of the paper is to comprehensively review the state-of-the-art TSR frameworks to highlights their strengths and weaknesses. Furthermore, the paper focuses on devising a detailed thematic taxonomy to classify existing literature that helps in understanding the underlying

issues and proof of concept. Moreover, the paper investigates critical aspects and related features of TSR framework and tools based on a set of defined parameters.

It also rigorously elaborated various testing domains and approaches followed by the extant TSR frame works. The results reveal that majority of TSR frameworks focused on randomized unit testing, and a considerable number of frameworks lacks in supporting multi-objective optimization problems. Moreover, there is no generalized framework, effective for testing applications developed in any programming domain. Conversely, Integer Linear Programming (ILP) based TSR frameworks provide an optimal solution for multi-objective optimization problems and improve execution time by running multiple ILP in parallel. The study concludes with new insights and provides an unbiased view of the state-of-the-art TSR frameworks. Finally, this paper present potential research issues for further investigation to anticipate efficient TSR frameworks.

Because many test cases have been removed, the testing information is also lost. Fault localization is a technique using testing information to locate the fault and widely used by programmers to debug programs, so it suffers from the side effects of test-suite reduction. How to reduce the test-suite size in software testing with the premise of retaining or improving fault-localization effectiveness has become the hot spot in the area of software debugging recently. In this paper, we propose a new approach that selectively keeps the limited redundant test cases in the reduced set; it makes the new reduced set relatively redundant compared to the original one, and we expect that it retains or even improves fault-localization effectiveness.

2.8 SIMILARITY BASED TEST CASE PRIORITIZATION

The paper [9] defines test suites often grow very large over many releases, such that it is impractical to re-execute all test cases within limited resources. Test case prioritization rearranges test cases to improve the effectiveness of testing. Code coverage has been widely used as criteria in test case prioritization. However, the simple way may not reveal some bugs, such that the fault detection rate decreases. In this paper, we use the ordered sequences of program entities to improve the effectiveness of test case prioritization. The execution frequency profiles of test cases are collected and transformed into the ordered sequences. We propose several novel similarity-based test case prioritization techniques based on the edit distances of ordered sequences. An empirical study of five open source programs was conducted. The experimental results show that our techniques can significantly increase the fault detection rate and be effective in detecting faults in loops. Moreover, our techniques are more cost-effective than the existing techniques.

Test case prioritization techniques can be used in many scenarios. In this paper focus on the regression testing scenario. Therefore, two cases need to be considered: general test case prioritization and version-specific test case prioritization. The purpose of test case prioritization is to increase the probability that test cases meet certain objectives when executed in a specific order. The main objective of the test case prioritization is to increase the fault detection rate. However, it is difficult to discover the fault detection information until the testing is finished. Hence, in practice, test case prioritization techniques rely on surrogates, hoping that early satisfying of these surrogates will lead to increasing the fault detection rate. Utilizing code coverage information as surrogates, coverage-based test case prioritization rearranges test cases in order to maximize code coverage as

early as possible. However, code coverage is not sufficient to guarantee a high fault detection rate in some cases.

Similarity-based techniques have been introduced to utilize the execution profiles of test cases. The purpose of similarity-based techniques is to maximize the diversity of selected test cases. The diversity of test cases is computed by a certain dissimilarity measure between each pair of test cases. Consequently, this will increase the chance of detecting faults as early as possible if we maximize the diversity of the test cases. Similarity-based techniques are mainly classified into two types: distribution-based and adaptive random testing (ART)-inspired. Distribution-based techniques cluster test cases according to their dissimilarities.

Clusters and isolated points can be used to guide test case selection and test case prioritization. The intuition behind the idea is that the test cases that find different faults belong to different clusters based on the similarity. ART-inspired test case prioritization is an extension of ART. Each time, a candidate set full of not-yet selected test cases is constructed and the test case farthest away from the prioritized test suite is selected as the next on eases.

In this method, proposed several novel similarity-based test case prioritization techniques based on the ordered sequences of program entities. The execution profile of each test case is first transformed into ordered sequences of program entities, sorted by the execution counts of each entity. Then, edit distance is used to calculate pair-wise distances. Two algorithms, FOS and GOS, are proposed to fulfill the test case prioritization techniques. It uses only prioritization technique. But combined approach gives best result.

2.9 SIMILARITY-BASED TEST SUITE REDUCTION

The paper [10] defines test suite reduction strategies aim to produce a smaller and representative suite that presents the same coverage as the original one but is more cost-effective. In the model based testing (MBT) context, reduction is crucial since automatic generation algorithms may blindly produce several similar test cases. In order to define the degree of similarity between test cases. However, there is still little or no knowledge on whether and how they influence on the performance of reduction strategies, particularly when considering MBT practices. This method investigates the effectiveness of distance functions in the scope of a MBT reduction strategy based on the similarity degree of test cases. It discusses six distance functions and applies them to three empirical studies. The first two studies are controlled experiments focusing on two real-world applications (and real faults) and ten synthetic specifications automatically generated from the configuration of each application (and faults randomly generated). In the third study, this method also applies the reduction strategy to two subsequent versions of an industrial application by considering real faults detected.

A similarity-based test suite reduction strategy inspired by the test selection strategy proposed by Cartaxo. The goal of the original selection strategy was to select a percentage of the test cases that are the most different ones based on the degree of similarity among them without the need to preserve test requirements coverage of the original suite. This paper used to refer the test suite reduction technique using similarity based approach. On the other hand, the reduction strategy aims to produce a subset from the original test suite that satisfies the same set of test requirements, by removing from the suite the most similar test cases while the reduced suite covers the requirements. In order to apply the reduction strategy, the following inputs are necessary,

1. Test suite the set of test cases to be reduced;

2. Test requirements the set of requirements that should be covered, defined by a satiability relation.

3. Similarity degree for all the pairs of test cases. The degree is measured by a Similarity matrix the matrix that presents the distance function.

This method discuss highlight that the number of paths with loops and the number of essential test cases in the specification configuration have also impact on the results of the reduction technique. When the number of paths with loops is high, probably the degree of redundancy in the test suite is high. Therefore, the reduction strategy can be more effective size and consequently less effective fault capability. When the number of essential test cases is high, observations are the opposite. Nevertheless, this is a behavior expected from the strategy of reduction based on similarity, as the average changes in rate are relatively similar when considering all function.

2.10 Limitations of Existing System

There are several techniques available for test case minimization and prioritization. Some of the test case minimization and prioritization techniques are explained and there are certain limitation for intend to apply requirement changes for performing the minimization and prioritization. Here regression technique any one method is used for optimal tests suite size. But this single method not given an effective result. We proposed, to combine both minimization and prioritization techniques used to produce the optimal test suite size.

CHAPTER 3

COVERAGE METRIC TECHNIQUES

3.1 WHITE BOX TESTING

White box testing is a testing technique that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing. White box testing involves looking at the structure of the code. When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification. And all internal components have been adequately exercised. The advantage of white box testing is Testing can be commenced at an earlier stage. Testing is more thorough, with the possibility of covering most paths. Spots the Dead Code or other issues with respect to best programming practices. White box testing is applicable for,

1) Unit testing
2) Integration testing
3) System testing

3.2 UNIT TESTING

Unit testing is one of the most popular software testing where Unit testing, a testing technique using which individual modules are tested to determine if there are any issues by the developer himself. It is concerned with functional correctness of the standalone modules. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc. In object-oriented

programming, the smallest unit is a method, which may belong to a base or super class, abstract class or derived/ child class. (Some treat a module of an application as a unit. This is to be discouraged as there will probably be many individual units within that module.) Unit testing frameworks, drivers, stubs, and mock or fake objects are used to assist in unit testing. The main aim is to isolate each unit of the system to identify, analyze and fix the defects. It benefit is, due to modular nature of the unit testing, we can test parts of the project without waiting for others to be completed.

3.3 CODE COVERAGE

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage. In most cases, code coverage system gathers information about the running program. It also combines that with source code information to generate a report about the test suite's code coverage. It helps to measure the efficiency of test implementation. It offers a quantitative measurement. It defines the degree to which the source code has been tested. In other words Code coverage is a measurement of how many lines/blocks/arcs of the code are executed while the automated tests are running. This code coverage technique used in united testing. Statement coverage, Block coverage, Control flow coverage, De- use coverage, Data flow coverage, Path coverage, Modified condition coverage

3.2.1 STATEMENT COVERAGE

Statement coverage is a white box test design technique which involves execution of all the executable statements in the source code at least once. It is

used to calculate and measure the number of statements in the source code which can be executed given the requirements.

Statement coverage= No. Of statements covered by test cases / Total of test cases×100

generally in any software, if we look at the source code, there will be a wide variety of elements like operators, functions, looping, exceptional handlers, etc. Based on the input to the program, some of the code statements may not be executed. The goal of Statement coverage is to cover all the possible paths, line, and statement in the code.

Sample code

```
1 READ A
2 READ B
3 C =A+ 2*B
4 IF C> 50 THEN
5 PRINT large C
6 ENDIF
```

Test cases

1. A=3 B=1
2. A=20 B=27

The first test case value testing the code completed 83% and the second test case testing the code completed 100%. This statement coverage is not suitable for true or false condition statement and it does not understand the logical operators.

3.3.2 BLOCK COVERAGE

The nature of the statement and block coverage looks somewhat same. The difference is that block coverage considers branched blocks of if/else, case

branches, wait, while, for etc. Block coverage defined as the ratio of number of block covered by test case and total number of block in the program.

Example

Program blocks are,

{1, 2, 3, 4, 5, 6}

One test case covered blocks

{1, 2, 5, 6}

Block coverage =>4/6=0.666

The block coverage of the program is 66%. The percentage of block coverage value is depending upon test cases.

3.3.3 CONTROL FLOW COVERAGE

The aim of this technique is to determine the execution order of statements or instructions of the program through a control structure. The control structure of a program is used to develop a test case for the program. In this technique, a particular part of a large program is selected by the tester to set the testing path. It is mostly used in unit testing. Test cases represented by the control graph of the program.

Control flow coverage depends upon the number of time true branch is executed and number of time false branch is executed in the program.

3.3.4 DEFINITION USE COVERAGE

Definition: data variables are defined, created and initialized, along with the allocation of the memory to that particular data objects.

Usage: declared data variables are used in the programming code in the forms as, the part of the predicate and in the in the computation form.

De-use defined as the ratio of number of test case cover the definition variable and use variable in the program and total number of the definition variable and use variable. Uses have two types. One is p-use and another one is c-use. The p-use is the predication variable like, if (i==0) here i value predicated. Then c-use is the computation variable like, i=n/2 here I value is calculated. De-use coverage percentage value is depend upon the test cases.

A Use-Definition Chain is a data structure that consists of a use, U, of a variable, and all the definitions, D, of that variable that can reach that use without any other intervening definitions. A definition can have many forms, but is generally taken to mean the assignment of some value to a variable (which is different from the use of the term that refers to the language construct involving a data type and allocating storage). A counterpart of a Use-Definition Chain is a Definition-use chain, which consists of a definition, D, of a variable and all the uses, U, reachable from that definition without any other intervening definition.

3.3.5 DATAFLOW COVERAGE

Data divergence is computed as the number of times test cases execute the conditional branches, loops and or blocks. Counts are used to indirectly represent the data values of variables in the software code under test.

Sample code

```
#include <iostream.h>
intmain ()
{
for (i=0;i<2;i++)  {
cout<< "Hello, World!";
   return 0;
```

In this program for loop is considered. Variable ‘i’ is two time running in the program. Value 0 and 1 is condition is true. Greater two conditions is false. This is used to calculate the dataflow coverage.

3.3.6 PATH COVERAGE

Path coverage testing is a specific kind of methodical, sequential testing in which each individual line of code is assessed**.** As a type of software testing, path coverage testing is in the category of technical test methods, rather than being part of an overarching strategy or "philosophy" of code. It is labor-intensive and is often reserved for specific vital sections of code.

The way that path coverage testing works is that the testers must look at each individual line of code that plays a role in a module and, for complete coverage; the testers must look at each possible scenario, so that all lines of code are covered. Experts generally consider path coverage testing to be a type of white box testing, which actually inspects the internal code of a program, rather just relying on external inputs and strategies that are considered black box testing, which do not consider internal code

Path testing is an approach to testing where; ensure that every path through a program has been executed at least once. This is one of the structural testing based on the method. Path testing fully depend on control flow graph (CFG).

3.3.7 MODIFIED CONDITION COVERAGE

The Modified Condition/Decision Coverage enhances the condition/decision coverage criteria by requiring that each condition be shown to independently affect the outcome of the decision. This kind of testing is performed on mission critical application which might lead to death, injury or monetary loss. Designing Modified Condition Coverage or Decision Coverage requires more thoughtful

selection of test cases which is carried out on a standalone module or integrated components. The characteristic of modified decision coverage is,

1) Every entry and exit point in the program has been invoked at least once.
2) Every decision has been tested for all the possible outcomes of the branch.
3) Every condition in a decision in the program has taken all possible outcomes at least once.
4) Every condition in a decision has been shown to independently affect that decision's outcome.

3.4 CODE COVERAGE METRIC

Code coverage metric is one of the measurements. This metric used to calculate the test cases similarities or diversities. Using this similarities and diversities we are identify the duplicate test cases. In this project four coverage metric are used. It's listed below,

1. Block coverage equivalence
2. Control flow coverage divergence
3. Def-use coverage equivalence
4. Data flow Coverage divergence.

The block coverage equivalence and def-use equivalence are help to calculate the equivalence signature value of the test cases. Control flow coverage divergence and data flow divergence are used to calculate the divergence signature value of the test cases.

3.4.1 BLOCK COVERAGE EQUIVALENCE

Block coverage equivalence measures block testing overlap between the two test cases of a test suite. The simple formula for block coverage equivalence is the

radio of adding the number of common blocks tested by test case pair and dividing this sum by the total number of unique blocks tested by both test case pair. This is used to calculate the similarity of the test case pair.

Block coverage equivalence, **M1 = X / Y**

X = No. Of common blocks between the Ti & Ti +1

Y = No. of Unique blocks between the Ti & Ti +1

3.4.2 CONTROL FLOW COVERAGE DIVERGENCE

Control flow divergence measures the dissimilarity of two test cases that test the same common block that have conditional path within them. For each test case that executes a block with a conditional branch a control flow (CF) value is calculated that provides a measurement for the number of times the test case takes each conditional branch. The branch is true and false.

$$\mathbf{CF_B = ((T - AVG\ (T\&F)) + (AVG\ (T\&F) - F)) / SUM\ (T\&F)}$$

T=No. of time true branch is executed

F=No. of time false branch is executed

Variance of Control flow value for a common block B of Ti & Ti +1

Variance,

$$\mathbf{\Delta B\ (Ti, Ti+1) = Square\ [CF_B\ (Ti) - M] + Square\ [CF_B\ (Ti+1) - M]}$$

Control flow divergence,

M2 = Sum of the variance of common blocks ($\sum \Delta B$ (Ti, Ti+1)) / No. of common blocks.

3.4.3 DEF-USE COVERAGE EQUIVALENCE

Def-use equivalence measures Def-use (definition or use) path testing overlap between two test cases in a test suit. A Def-use path is a logic execution sequence in a block that defines and uses a variable. The de-use paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used**.** It is used to calculate the similarity of test cases.

Def- use equivalence, **M3 = X / Y**

X = No. of common def-use chain tested by test case pairs Ti & Ti+1

Y = No. of unique def-use chain tested by test case pairs Ti & Ti +1

3.4.4 DATA FLOW COVERAGE DIVERGENCE

Data divergence measures the diversity of test cases with respect to the data values the test cases use for code variables. Data divergence is computed as the similarity of the number of times two test cases execute the same conditional branches, loops or blocks.

Data flow divergence, **DF (Ti, B) = (T – F) /2**

T = No. of times true branch is executed

F = No. of times false branch is executed.

Variance of data flow divergence,

ΔB (Ti, Ti+1) = Square (DF_B (Ti) – M) + Square (DF_B (Ti+1) – M).

Data flow divergence,

M4 = Sum of the variance of common blocks ($\sum\Delta$B (Ti, Ti+1)) / No. of common blocks.

CHAPTER 4

SYSTEM ANALYSIS

4.1 DATA FLOW DIAGRAM

A data flow diagram (DFD) maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination. Data flowcharts can range from simple, even hand-drawn process overviews, to in-depth, multi-level DFDs that dig progressively deeper into how the data is handled. They can be used to analyze an existing system or model a new one. Like all the best diagrams and charts, a DFD can often visually "say" things that would be hard to explain in words, and they work for both technical and nontechnical audiences, from developer. That's why DFDs remain so popular after all these years. While they work well for data flow software and systems, they are less applicable nowadays to visualizing interactive, real-time or database-oriented software or systems.

They based it on the "data flow graph" computation models by David Martin and Gerald Estrin. The structured design concept took off in the software engineering field, and the DFD method took off with it. It became more popular in business circles, as it was applied to business analysis, than in academic circles.

Also contributing were two related concepts:

1. Object Oriented Analysis and Design (OOAD), put forth by Yourdon and Peter Coad to analyze and design an application or system.
2. Structured Systems Analysis and Design Method (SSADM), a waterfall method to analyze and design information systems. This rigorous

documentation approach contrasts with modern agile approaches such as Scrum and Dynamic Systems Development Method (DSDM.)

1. **External entity:** an outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system. They might be an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.
2. **Process:** any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules. A short label is used to describe the process, such as "Submit payment."
3. **Data store:** files or repositories that hold information for later use, such as a database table or a membership form. Each data store receives a simple label, such as "Orders."
4. **Data flow:** the route that data takes between the external entities, processes and data stores. It portrays the interface between the other components and is shown with arrows, typically labeled with a short data name, like "Billing details.

A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond. The necessary level of detail depends on the scope of what you are trying to accomplish.

LEVEL 0

DFD Level 0 is also called a Context Diagram. It's a basic overview of the whole system or process being analyzed or modeled. It's designed to be an at-a-

glance view, showing the system as a single high-level process, with its relationship to external entities. It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers. The Context level diagram in fig 4.1.

Fig 4.1 Context Level Diagram

LEVEL 1

DFD Level 1 provides a more detailed breakout of pieces of the Context Level Diagram. You will highlight the main functions carried out by the system, as you break down the high-level process of the Context Diagram into its sub processes. The diagram for generating optimal test suite size in fig 4.2.

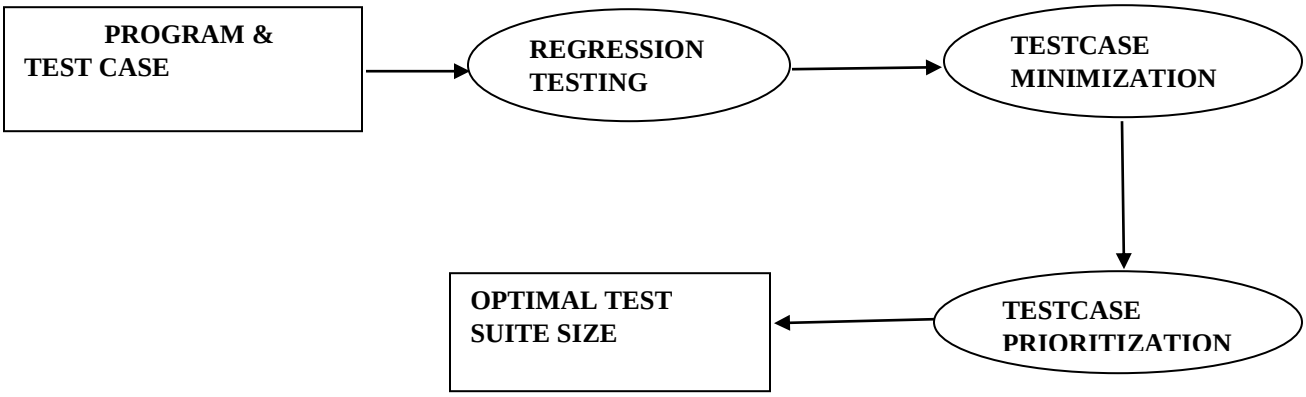

Fig 4.2 Level 1- Generating optimal test suite size

The context level diagram split into sub process. The regression testing split into test case minimization and test case prioritization.

LEVEL 2

DFD Level 2 then goes one step deeper into parts of Level 1. It may require more text to reach the necessary level of detail about the system's functioning. This is the final level of data flow diagram of our process, here the process are deeply explained. The Effective minimization and prioritization in fig 4.3.

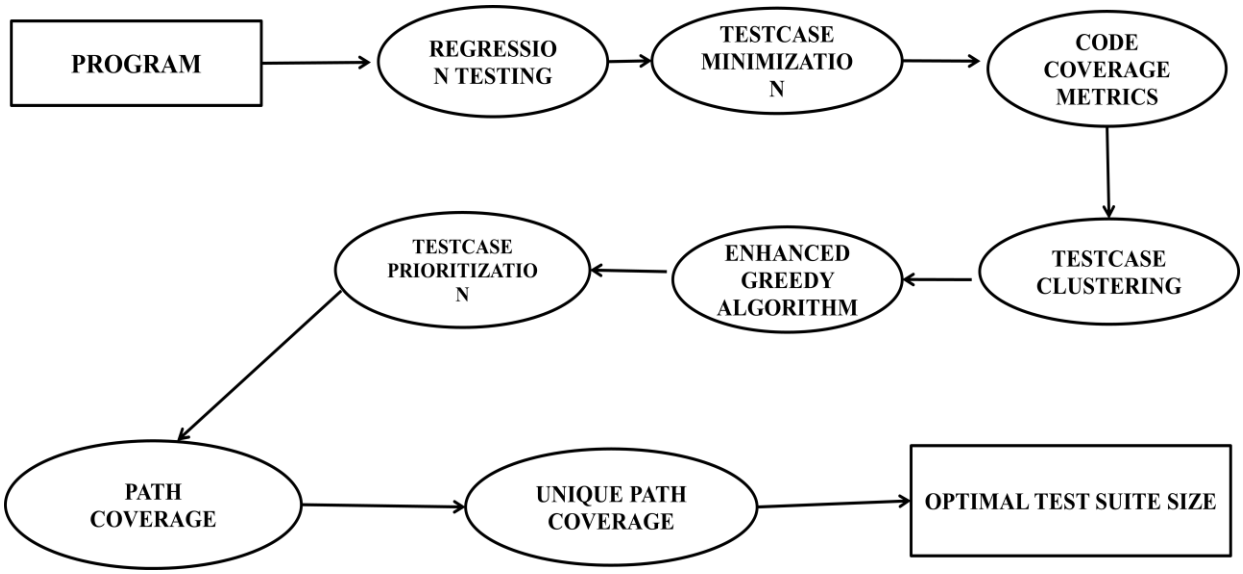

Fig 4.3 Level 2 -Process of minimization and prioritization

4.2 SYSTEM ARCHITECTURE

A system architecture is the conceptual model that defines the structure, behavior, and more views of a system. An architecture description is a formal description and representation of a system, organized in a way that supports reasoning about the structures and behaviors of the system. A system architecture can consist of system components and the sub-systems developed, that will work together to implement the overall system. There have been efforts to formalize languages to describe system architecture; collectively these are called architecture description languages. The architectural diagram for our approach is given as follows in fig 4.4.

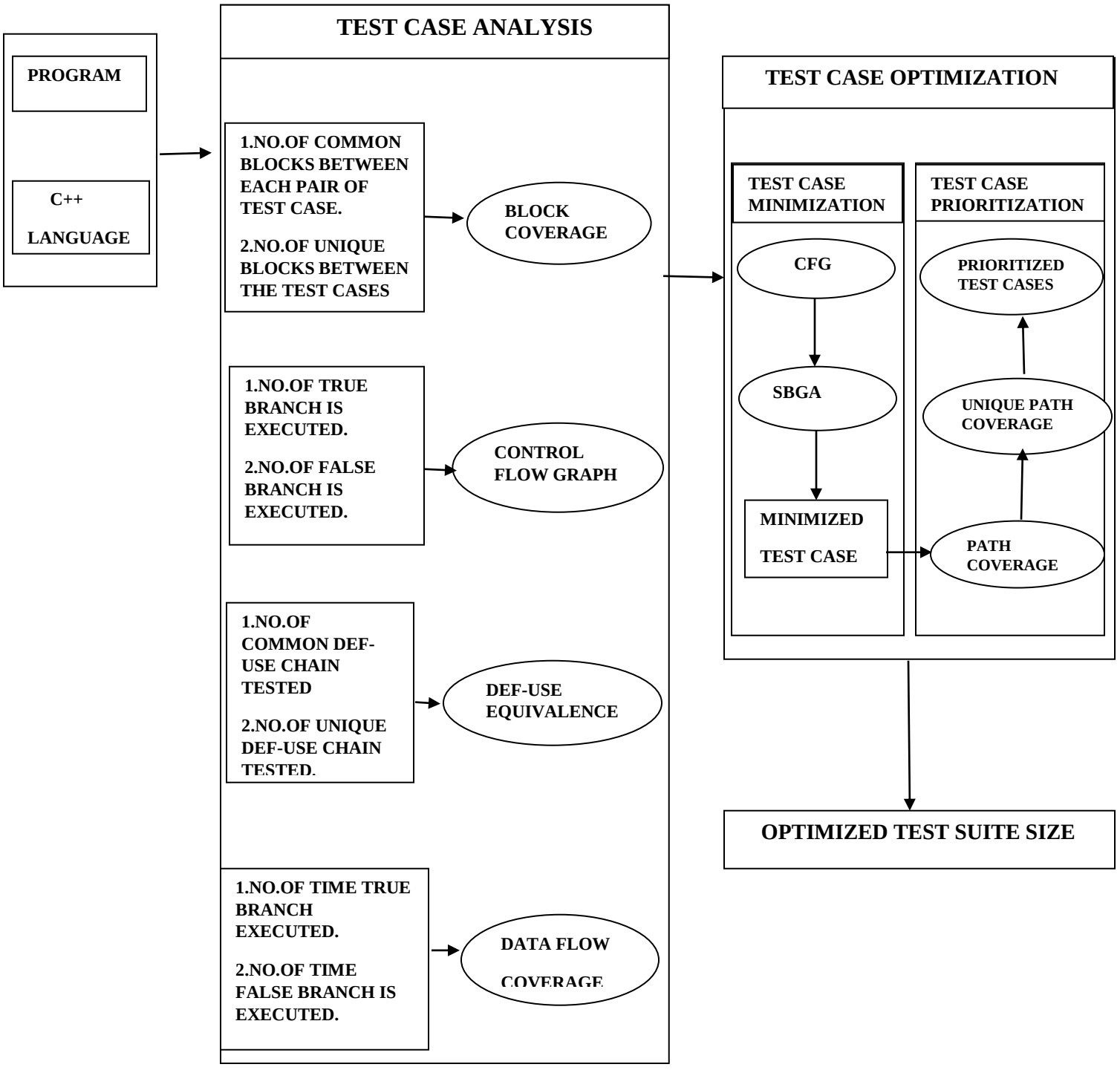

Fig 4.4 Architecture for optimal test suite size

Has shown in the above diagram, Software regression testing is hard for a single test case to satisfy the coverage of entirely given test requirements, because of the extensive use of testing to measure the quality of the software, one of the challenges faced by organizations is the test suite optimization. In testing the requirements are gathered from Software Requirement and Specification (SRS). Once a set of requirements is found, a test set is generated to fulfill the requirements manually or automatically. A test case is a collection of input data and expected output data, which are mainly created to evaluate a particular software function by executing the software against these.

Some of the test cases in a constructed test suite may become duplicate, because the test cases created specifically for a testing requirement or testing criteria may also satisfy other requirements, and a requirement may still satisfy by some of the proper subsets of the test suite. The prime objective is to remove the duplicate test cases and extract the essential or diverse test cases to generate the optimal test suite.

The different approaches such as test case selection, minimization and prioritization techniques. To address this problem, a number of techniques and framework have been proposed to make the reduction process more effective which based on test case classification according to similarity degree measured by a distance function. Diversity and Similarity based test case selection and prioritization are one of the new approaches with desired output. But the above techniques did not provide any systematic approach for prioritization and minimization both. So, the study suggests that using the greedy technique with similarity based approach could be a better option for effective testing.

The proposed a similarity based greedy approach for test suite reduction. The main idea is to analysis the similarity degree among test case pairs and systematically removes them by applying enhanced greedy algorithm while maintaining test requirements coverage. Here we used the coverage metrics i.e. Block, Control flow, Def-use and Data flow coverage to compute the similarity values for each test case pairs. The other phase of prioritization is proposed to derive the Similarity Based Greedy Approach (SBGA) by using their unique path coverage information.

CHAPTER 5

PROPOSED TECHNIQUES

To optimize the test cases we have proposed two approaches, one is signature and another one is Similarity Based Greedy Approach (SBGA). Program for the above process have been shown in Table 5.1. The test case for the program shown in table 5.2.

Find the no. of days in a month, using the given month and year

```
int main()   {
int days, month, year;
cout<< " \n enter the month:" ;
cin >> month;
while (month > 12 || month < 1)  {
cout << " \n invalid month " ;
cout << " \n enter the month :" ;
cin >> month ;  }
cout<< " \n enter the year :" ;
cin>> year;
if ( month ==2)    {
If ((year % 400 ==0) || (year % 4 ==0 && year % 100 !=0))
days = 29;else
days =28;    }
else if ( month ==1 || month == 3 || month == 5 || month ==7 || month == 8 || month == 10 || month == 12 )
days = 31;else
days = 30;
cout<< " no. of days : " << days;
return 0;}
```

Fig 5.1 sample program

Test cases	Test cases input		Expected Output (Days)
	Month	Year	
T1	1	2018	31
T2	4	2017	30
T3	2	3000	28
T4	2	1997	28
T5	2	2000	29
T6	2	2016	29
T7	0, 13, 6	2000	30

Table 5.2 Test cases for the sample program

5.1 IDENTIFY THE DUPLICATE TEST CASES

1. Calculate the code coverage metrics values. Such as, block coverage, control flow coverage, def-use equivalence, and data flow coverage.
2. Calculate equivalence signature using block coverage.
3. Calculate divergence signature using control flow coverage, Data flow coverage.
4. After calculating equivalence signature and divergence signature the threshold value will be calculated.

5.1.1 COVERAGE METRIC VALUES

Test cases		Coverage metrics values of Test cases						
		T1	T2	T3	T4	T5	T6	T7
T1	M1	0.00	0.66	0.66	0.66	0.66	0.66	0.57
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.88
	M3	0.00	0.75	0.55	0.55	0.55	0.55	0.75
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.50
T2	M1	0.66	0.00	0.66	0.66	0.66	0.66	0.83
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.88
	M3	0.75	0.00	0.55	0.55	0.55	0.55	0.75
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.50
T3	M1	0.66	0.66	0.00	1.00	0.66	0.66	0.57
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.88
	M3	0.55	0.55	0.00	1.00	0.75	0.75	0.54
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.50
T4	M1	0.66	0.66	1.00	0.00	0.66	0.66	0.57
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.88
	M3	0.55	0.55	1.00	0.00	0.75	0.75	0.54
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.50
T5	M1	0.66	0.66	0.66	0.66	0.00	1.00	0.57
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.88
	M3	0.55	0.55	0.75	0.75	0.00	1.00	0.54
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.50
T6	M1	0.66	0.66	0.66	0.66	1.00	0.00	0.57
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.88
	M3	0.55	0.55	0.75	0.75	1.00	0.00	0.54
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.50
T7	M1	0.83	0.57	0.57	0.57	0.57	0.57	0.00
	M2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	M3	0.75	0.75	0.54	0.54	0.54	0.54	0.00
	M4	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.3 Coverage metric values

The above Table 5.3shows the coverage metrics values. This value calculated using coverage metrics formulas.

5.1.2 EQUIVALENCE SIGNATURE VALUE

Each test case signature is an aggregate quantifiable metric that is used to identify the amount of similarity or dissimilarity of the test cases of a test case pair. A signature is a weighted average of a subset of the metric generated for a test case pair.

Thus, once a set of metric value are established for a test case pair each metric is weighted, a subset of the weighted metrics are summed, and the result is divided by the number of added metrics, to define signature for the test case pair. Each metric is given equal weight, or importance, and thus the weight assigned each metric is one. A metric can be disabled by assigning it a weight of zero.

Equivalence signature= (P1M1+P3M3)/2

Where, P1 = P3 = 1 (Px is the weight for the Xth metric),

Mx is the xth metric value for test case pair,

M1=block Coverage Equivalence Metric Value

M3 = DU equivalence metric value.

Example

M1=0.66

M3=0.75

Equivalence signature= (0.66+0.75)/2

= 0.70

It is the equivalence signature value of T1 and T2. This model is followed each test case pairs.

5.1.3 DIVERGENCE SIGNATURE VALUE

After calculating these metrics values, the signature values are calculated using the test case comparison metrics algorithm. Signature values for each of the test cases are calculated according to the above formulae. For each test case pair calculated equivalence and divergence signature values.

Divergence signature=(P2M2+P4M4)/2

Where, P2 = P4 = 1 (Px is the weight for the Xth metric),

Mx is the xth metric value for test case pair,

M2 = Control Flow Divergence metric value,

M4 = Data Flow Divergence metric value.

Example

M2=0.88M4=0.50

Equivalence signature= (0.88+0.50)/2

= 0.69

It is the divergence signature value of T1 and T7. This model is followed each test case pairs.

5.1.4 SIGNATURE VALUE PAIRS

After making a group of similar and diverse test cases, the threshold values of divergence and equivalence as 0.05 and 0.90 are assumed. Only those test cases having divergence value less than (0.05) and equivalence value greater than (0.90) comes under the similar or duplicate group and the remaining test cases are selected for diverse group. The signature value pairs have been shown in table 5.4.

Test cases	Distance	Signature values						
		T1	T2	T3	T4	T5	T6	T7
T1	Div.	-	0.00	0.00	0.00	0.00	0.00	0.69
	Comm.	-	0.70	0.60	0.60	0.60	0.60	0.66
T2	Div.	0.00	-	0.00	0.00	0.00	0.00	0.69
	Comm.	0.70	-	0.60	0.60	0.60	0.60	0.66
T3	Div.	0.00	0.00	-	0.00	0.00	0.00	0.69
	Comm.	0.60	0.60	-	1.00	0.70	0.70	0.55
T5	Div.	0.00	0.00	0.00	0.00	-	0.00	069
	Comm.	0.60	0.60	0.70	0.70	-	1.00	0.55
T7	Div.	0.69	0.69	0.69	0.69	0.69	0.69	-
	Comm.	0.66	0.66	0.55	0.55	0.55	0.55	-

Table 5.4 Signature value pairs

Using threshold values T3& T4 and T5& T6 values are similar to each other. So, any one test case will be randomly removed.

5.2 SIMILARITY BASED GREEDY APPROACH

Similarity Based Greedy Approach (SBGA) is implemented using java coding; this similarity based greedy procedure used for test case minimization and test case prioritization. Similarity based greedy approach, required input is path of the test case and output is minimized test case, also prioritized test case. The test cases path has been calculated using control flow graph. The path values are implemented using SBGA approach.

PROCEDURE

1. Declare the total no. of Test cases and its elements.
2. Calculate the each Testcase elements count.
3. Declare the total no .of path.
4. Minimum elements of Test case would be compared with total no .of path.
5. These comparisons usedto find the duplicate Test case. The result of comparison considered as‘s’ elements.
6. S – Elementsintersected with other test cases.
7. Calculate the countof each intersected values.
8. Find the maximum ofintersected test case.
9. Combine the elementsof minimum elementtest caseand maximum intersected test case.
10. Follow step 4 up to step 8 until the entire path is covered.
11. The minimized test case unique path is calculated.
12. Calculate the maximum unique path with high to low order test case value.

The procedure used to calculate the minimized path coverage of the test cases. This Similarity Based Greedy Approach used to find out the optimal size of the test cases.

5.2.1 CONTROL FLOW GRAPH

In a control-flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

The CFG can thus be obtained, at least conceptually, by starting from the program's (full) flow graph-i.e. the graph in which every node represents an individual instruction and performing an edge contraction for every edge that falsifies the predicate above, i.e. contracting every edge whose source has a single exit and whose destination has a single entry. This contraction-based algorithm is of no practical importance, except as a visualization aid for understanding the CFG construction, because the CFG can be more efficiently constructed directly from the program by scanning it for basic blocks.

Because of its construction procedure, in a CFG, every edge A→B has the property that,

out degree(A) > 1 or in degree(B) > 1 (or both).

For further approach Control flow graph is used to perform the minimization and prioritization. We uses our sample program to draw the control flow graph to specify the traveling path of the each remaining test cases for further details. The control flow graph for the program shown in fig 5.1.

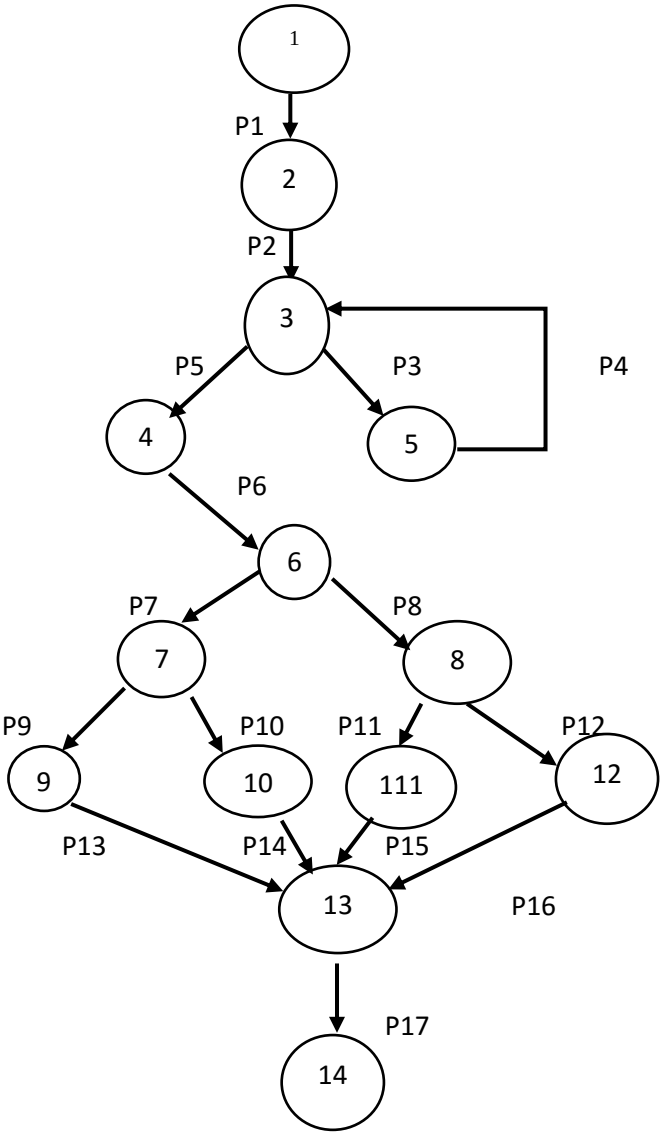

Fig 5.1 control flow graph for the program

The implementation of control flow graph have been derived in visustin tool shown in Fig 5.2.

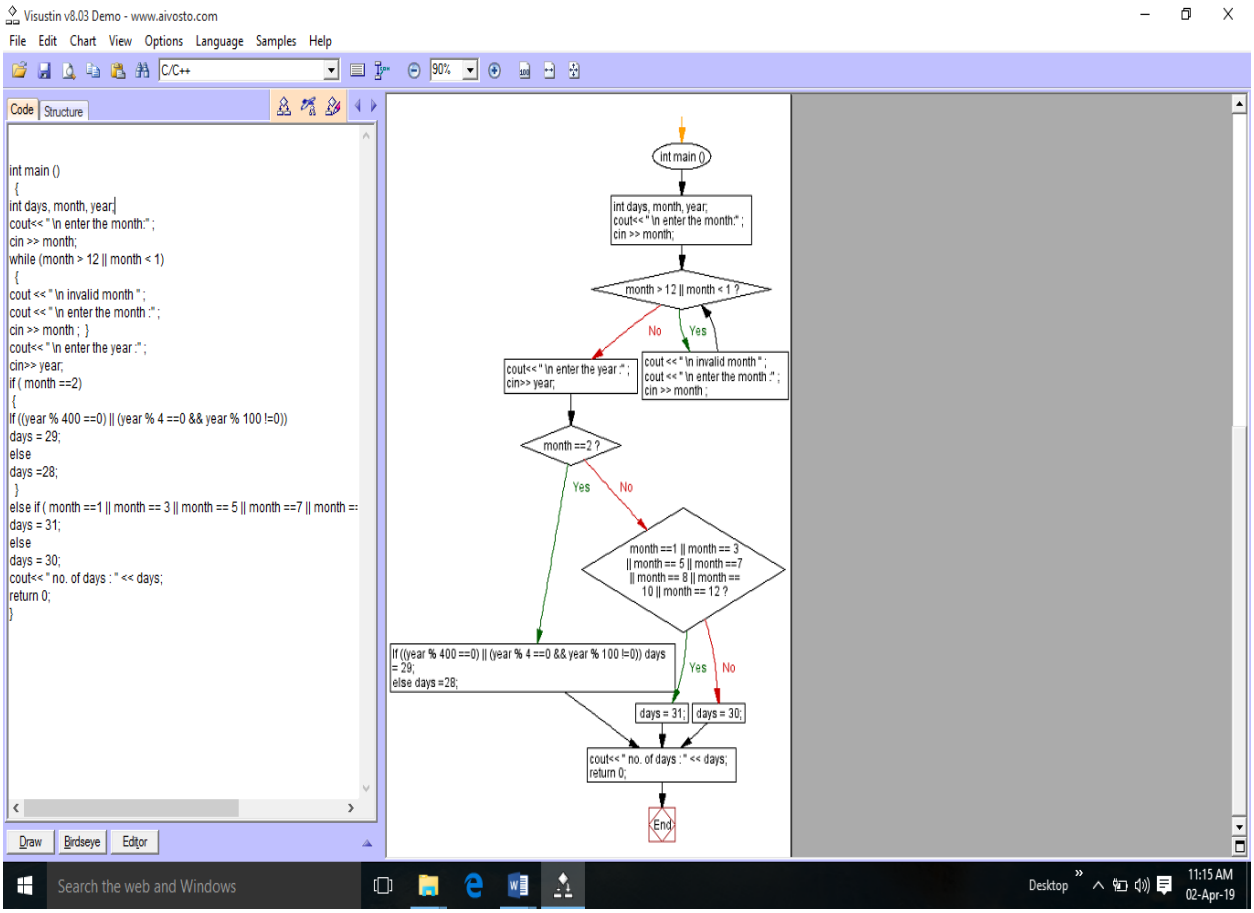

Fig 5.2 CFG in visustin tool

Example: find the days given a month program test case traveling path are listed below,

T1= {1, 2, 5, 6, 8, 11, 15, 17}

` T2= {1, 2, 5, 6, 8, 12, 16, 17}

T3= {1, 2, 5, 6, 7, 10, 14, 17}

T4= {1, 2, 5, 6, 7, 10, 14,17}

T5= {1, 2, 5, 6, 7, 9, 13, 17}

T6= {1, 2, 5, 6, 7, 9, 13, 17}

T7= {1, 2, 3, 4, 5, 6, 8, 12, 16, 17}

5.2.2 GENARATE MINIMIZED TEST CASES

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. Minimized test case has been shown in table 5.5.

Test cases	Path Coverage	No. of path covered by test cases
T1	{ 1,2,5,6,8,11,15,17}	8
T3	{1,2,5,6,7,10,14,17}	8
T5	{1,2,5,6,7,9,13,17}	8
T7	{1,2,3,4,5,6,8,12,16,17}	10

Table 5.5 minimized test cases

5.2.3 PRIORITISE MINIMIZED TEST CASES

The above table shows the minimized test cases. The each test cases travel through the control flow graph for calculate the path. The path of the each test case has been shown in table 5.6.

Test cases	Path Coverage (p)																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T1	1	1	0	0	1	1	0	1	0	0	1	0	0	0	1	0	1
T2	1	1	0	0	1	1	0	1	0	0	0	1	0	0	0	1	1
T3	1	1	0	0	1	1	1	0	0	1	0	0	0	1	0	0	1
T4	1	1	0	0	1	1	1	0	0	1	0	0	0	1	0	0	1
T5	1	1	0	0	1	1	1	0	1	0	0	0	1	0	0	0	1
T6	1	1	0	0	1	1	1	0	1	0	0	0	1	0	0	0	1
T7	1	1	1	1	1	1	0	1	0	0	0	1	0	0	0	1	1

Table 5.6 Path coverage for each test case

As shown in the above diagram the path coverage will calculate for the each test cases, which test case having more number of unique path is calculated. The unique path test cases will ordered as high to low .Unique path coverage in table 5.7.

Test cases	Path Coverage	No. of path covered by test cases	Unique path coverage
T7	{ 1,2,3,4,5,6,8,12,16,17}	10	4{3,4,12,16}
T1	{1,2,5,6,8,11,15,17}	8	2{11.15}
T3	{1,2,5,6,7,9,10,14,17}	8	2{10,14}
T5	{1,2,5,6,7,9,13,17}	8	2{9,13}

Table 5.7 Optimal test suite

Result of optimal test suite size in fig 5.3.

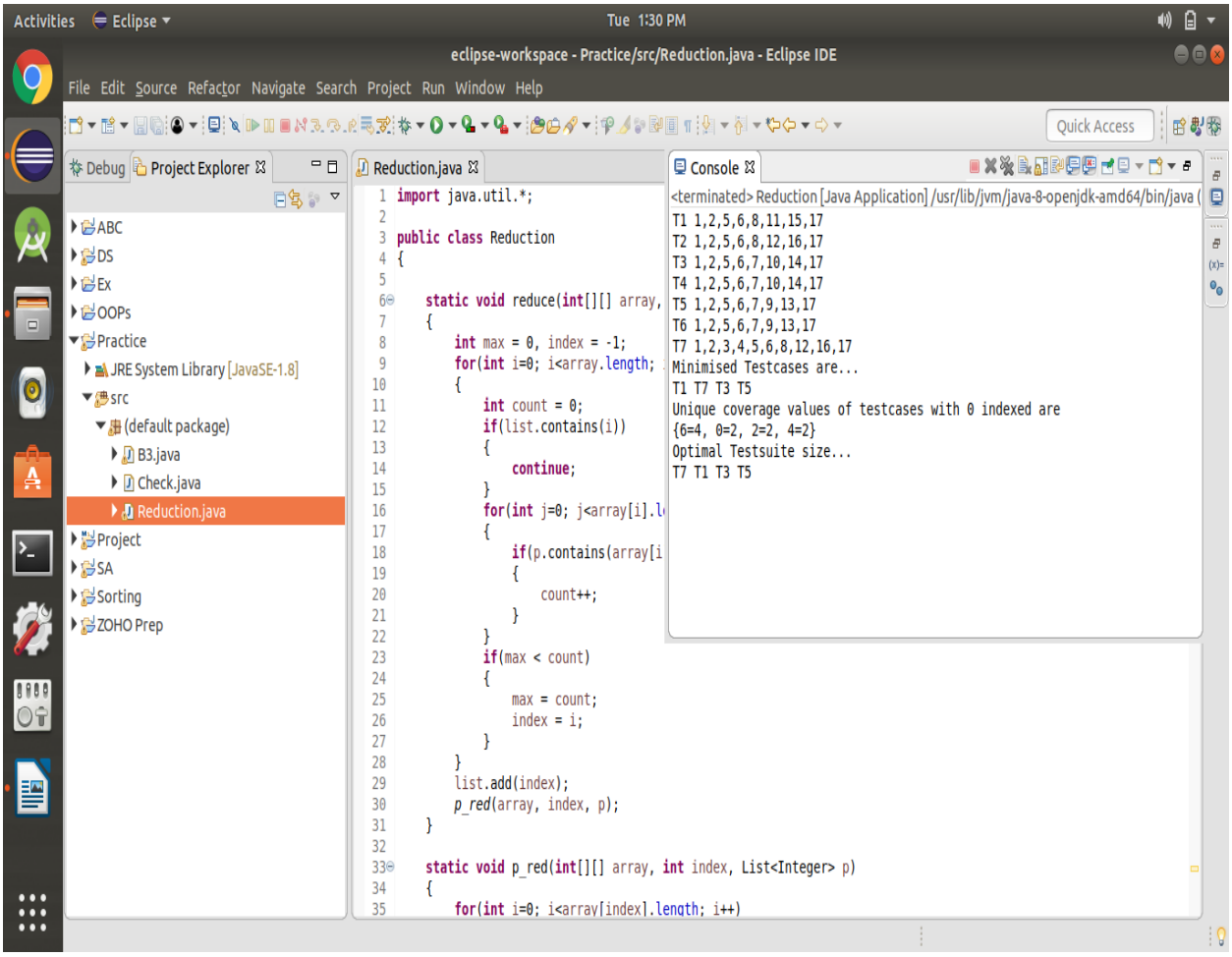

Fig 5.3 Result of optimized test suite size

5.3 PERFORMANCE METRICS

The following metrics are used for performance evaluation of the proposed and state-of-the-art algorithms and are described as follows: The first test metric implies the percentage reduction in test suite i.e. Suite Size Reduction (SSR). This is calculated as in Eq. (1)

$$SSR = 100 \times (1 - T_{rs} / T) \qquad (1)$$

On the other hand, second metric Fault detection loss (FDL) percentage signifies the total number of faults revealed by the minimized test suite. It is calculated as in Eq. (2)

$$FDL = 100 \times (1 - F_m / F) \qquad (2)$$

Where F represents the total number of distinct faults revealed by original test suite and Fm is the number of distinct faults exposed by minimized test suite.

CHAPTER 6

DISCUSSION ON EXPERIMENTAL RESULTS

6.1 EFFECTIVENESS USING COMBINED APPROACH

To validate proposed approach, we have considered five subject programs. That is,

1) Check the number is positive, negative or zero.
2) Leap year or not.
3) Prime or not prime.
4) Vowel or consonant.
5) Find the days in a given month.

Code coverage metric are used to identify the duplicate test case and also test the program for fault detection.

The control flow graph was implemented using visustin tool. The steps are followed in visustin tool to draw the control flow graph. By selecting the format of the code that is C/C++ and then select the go button to import the code. Using the menu for conversion of program into CFG. It will generate the separate graphs for separate sub-functions.

The control flow graph denote the edge and nodes, then calculate the test case pass through the node numbers, this will called as path coverage of the test cases.

Test case minimization and test case prioritization combined to obtain the optimal test suite solution. Here, two performance metrics is used. Suite Size Reduction (SSR) metric used to calculate the percentage of reduced test case. Fault Detection loss (FDL) metric used to perform signifies the minimized test suite. Fault Detection Loss for the Prime or not prime, the performance metric shown in Table 6.1.

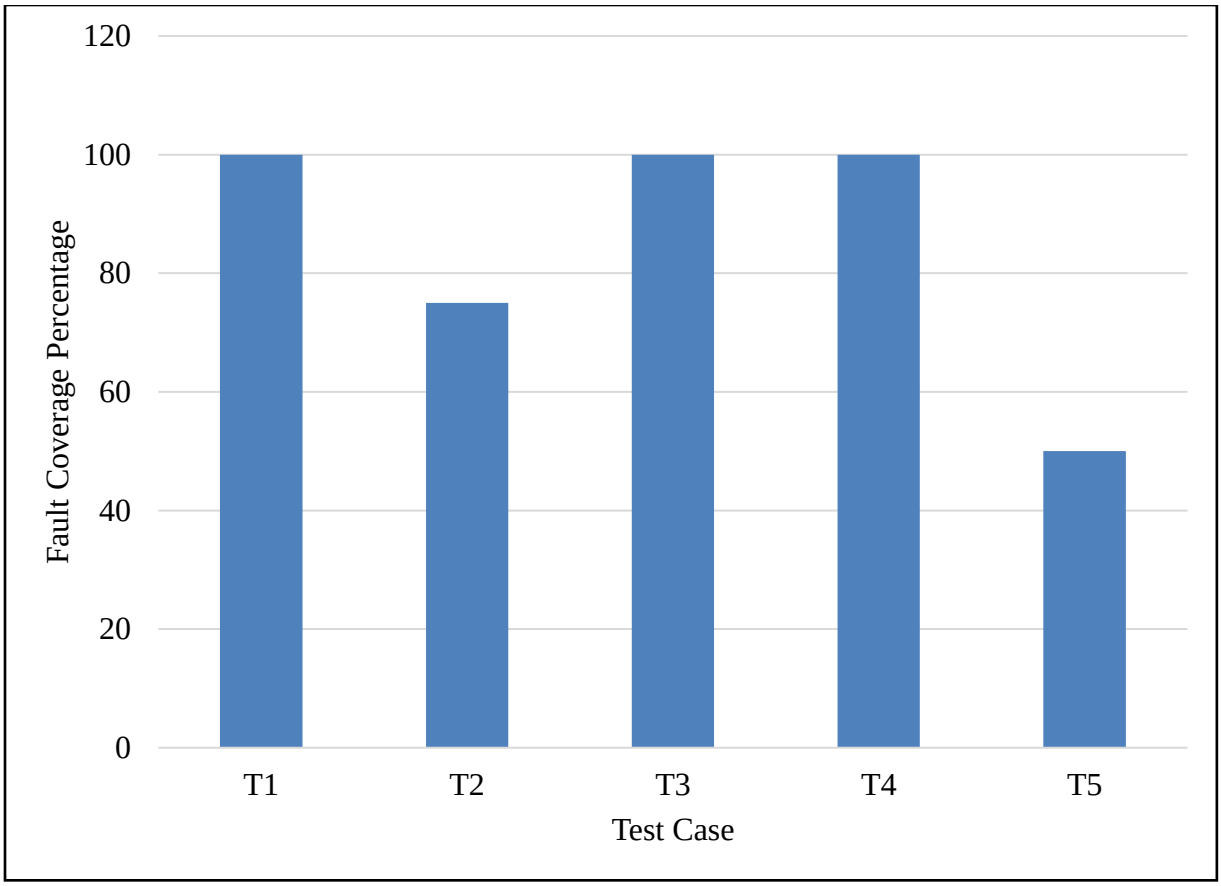

Fig 6.1 Performance metric graph

6.2 COMPARATIVE STUDY

The comparison of results of our proposed approach with existing approach using SBGA for different selected programs is given in Table 6.1 and the graph for the performance metric in Fig 6.2.

Program	**Original test case size**	**Reduced test case size**
Check the number is positive, negative or zero	**7**	**3**
Leap year or not	**6**	**4**

Prime or not	**6**	**3**
Vowel or Consonant	**6**	**3**
Find the days given month & year.	**7**	**4**

Table 6.1 Proposed results of five subject programs

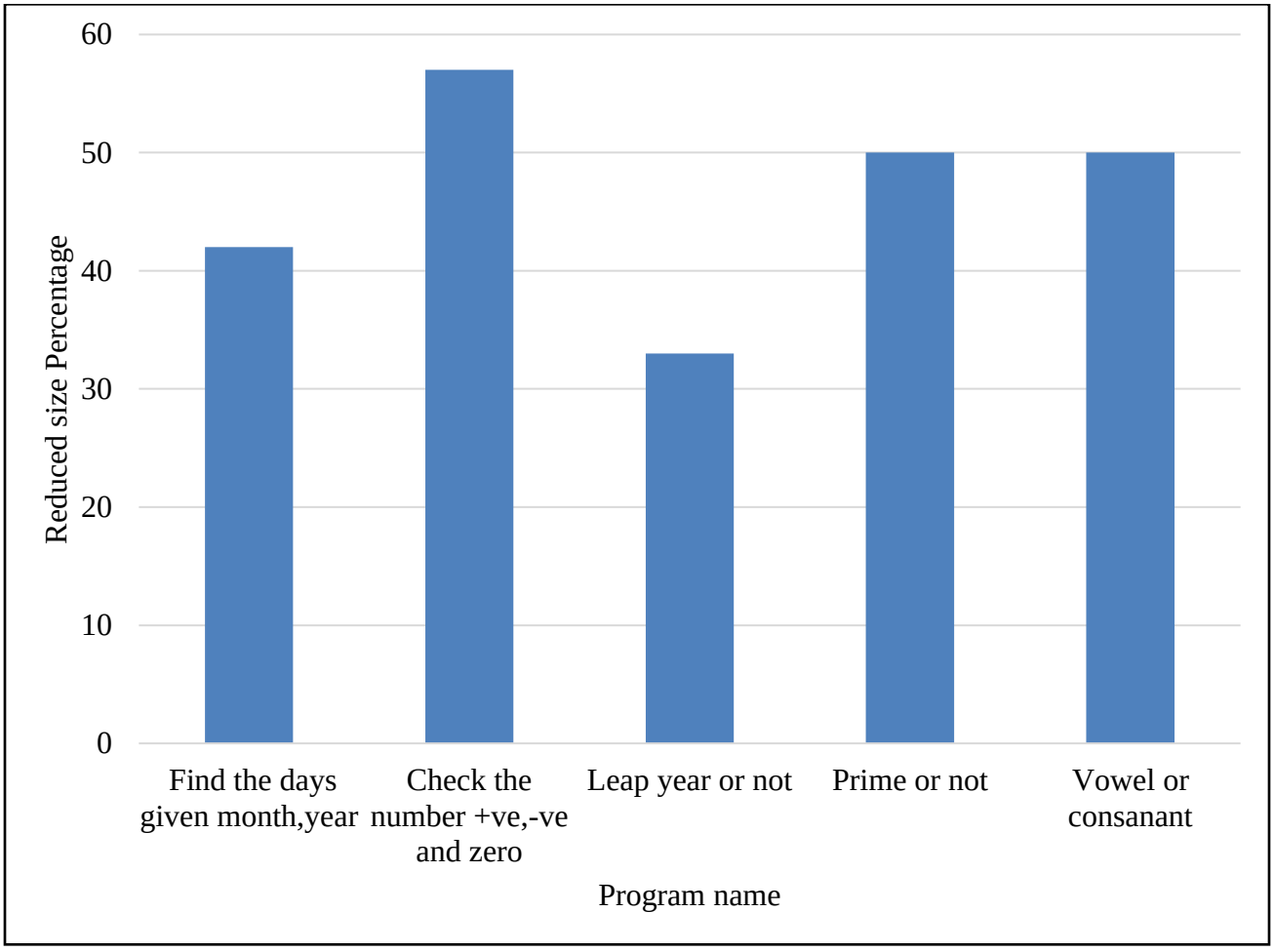

Fig 6.2 Suite size reduction for five subject programs

CHAPTER 7

7.1 CONCLUSION

In this project we address the optimization of test suite size with five subject programs. By using this different level of coverage metrics to find the signature values of the test case pairs and duplicate test cases also removed. Similarity Based Greedy Approach (SBGA) used for minimization and prioritization. The proposed approach can be useful to find the fault detection effectiveness is more important as compared to code coverage. This project also presents the results of experimental study of sample program (find the days in a given month). To evaluate the effectiveness of the proposed work two performance metrics were used. Moreover, experimental evaluation of other similarity functions with a different combination of coverage criteria needs to be conducted for getting more optimal test suite size.

7.2 FUTURE WORK

In future, we will considering the various level of complex program for test case minimization and test case prioritization. Data mining concept clustering technique is applied for identify the similarity of the test cases in complex program.

REFERENCES

[1] S.U.R. Khan, A. Nadeem, and A. Awais, “TestFilter: A Statement Coverage based TestCase Reduction Technique”, In: Proc. of 10th IEEE International Multitopic Conference (INMIC’06), pp. 275-280, 2006.

[2] S. Singh, C. Sharma, and U. Singh, “A Simple Technique to Find Diverse Test Cases”, In: Proc. of 9th International ICST Conference on Heterogeneous Networking for Quality, Reliability, Security, and Robustness, pp. 1-4, 2013.

[3] R. Gupta and M. L. Soffa, "Employing static information in the generation of test case", Software Testing, Verification and Reliability Vol.3, No.1, pp. 29-48, 1993.

[4] D. Binkley, “Semantics guided regression test cost reduction”, IEEE Transactions on Software Engineering, Vol.23, No.8, pp.498– 516, 1997.

[5] H. Zhong, L. Zhang, and H. Mei, “An experimental study of four typical test suite reduction techniques”, Information and Software Technology, Vol.50, No. 6, pp. 534– 546, 2008.

[6] A. Ilkhani and G. Abaee, “Extracting test cases by using data mining; reducing the cost of testing”, In: Proc. of International Conference on Computer Information Systems and Industrial Management Applications, pp.730– 737, 2011.

[7] 8. J. R. Horgan and S. London, “A data flow coverage testing tool for C”, In: Proc. of the Second IEEE Symposium on Assessment of Quality Software Development Tools, pp. 2-10, 1992.

[8] H. S. Chae, G. Woo, T. Y. Kim, J. H. Bae, and W. Y. Kim, “An automated approach to reducing test suites for testing retargeted C compilers for embedded systems”, Journal of Systems and Software, Vol.84, No.12, pp. 2053-2064, 2011.

[9] X. Zhang, Q. Gu, X. Chen, J. QI, and D. Chen, “A study of relative redundancy in test-suite reduction while retaining or improving faultlocalization effectiveness”, In: Proc. of the ACM Symposium on Applied Computing (SAC'10), Sierre, Switzerland, pp. 2229-2236, 2010.

[10] V. Chvatal, “A greedy heuristic for the setcovering problem”, Mathematics of operations research, Vol.4, No.3, pp.233-235, 1979.

[11] T. Y. Chen and M. F. Lau, “A simulation study on some heuristics for test suite reduction”, Information and Software Technology, Vol.40, No.13, pp. 777-787, 1998.

[12] T. Y. Chen and M. F. Lau, “A new heuristic for test suite reduction”, Information and Software Technology, Vol.40, No.(5-6), pp.347-354, 1998.

[13] M. J. Harrold, R. Gupta, and M. L. Soffa, “A methodology for controlling the size of a test suite”, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol.2, No.3, pp. 270-285, 1993.

[14] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness”, In: Proc. of the 36th International Conference on Software Engineering, pp. 435-445, 2014.

[15] C. Sharma and S. Singh, “Mechanism for identification of duplicate test cases”, In: Proc. of Recent Advances and Innovations in Engineering (ICRAIE), pp. 1-5, 2014.

[16] S. Singh and R. Shree, “A combined approach to optimize the test suite size in regression testing”, CSI transactions on ICT, Vol.4, No. (2-4), pp. 73-8, 2016.

[17] A. E. V. B. Coutinho, E. G. Cartaxo, and P. D. L. Machado, “Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing”, Software Quality Journal,Vol.24, No.2, pp.407-445, 2016.

[18] H. Hemmati, A. Arcuri, and L. C. Briand, “Achieving scalable model-based testing through test case diversity”, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol.22, No.1, pp. 1–42, 2013.

[19] R.Wang, S.Jiang and D.Chen, ”Similarity-based regression test case prioritization”, In: Proc. of the 27th International Conference on Software Engineering and Knowledge Engineering, pp. 358-363, 2015.

[20] S. R. Sugave, S. H. Patil, and B. E. Reddy, "A Cost-Aware Test Suite Minimization Approach Using TAP Measure and Greedy Search Algorithm", International Journal of Intelligent Engineering and Systems, Vol.10, No.4, pp.6069, 2017.

[21] M. Tulasiraman and V. Kalimuthu, “Cost Cognizant History Based Prioritization of Test Case for Regression Testing Using Immune Algorithm”, International Journal of Intelligent Engineering and Systems, Vol.11, No.1, pp.221-228, 2018.

[22] S. Yoo and M. Harman, “Regression testing minimization, selection, and prioritization: asurvey”, Softw. Test., Verif. Reliable, Vol.22, No.2, pp. 67–120, 2012.

[23] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, and V. Chang, "A survey on Test Suite Reduction frameworks and tools", International Journal of Information Management, Vol.36, No.6, pp.963-975, 2016.

[24] H. Hemmati and L. Briand, “An industrial investigation of similarity measures for modelbased test case selection”, In: Proc. of IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp. 141-150, 2010.

[25] E. G. Cartaxo, P. D. Machado, and F. G. O. Neto, “On the use of a similarity function for test case selection in the context of modelbased testing”, Software Testing, Verification and Reliability, Vol.21, No.2, pp.75-100, 2011.

[26] A. E. V. B. Coutinho, E. G. Cartaxo, and P. D. L. Machado, "Test suite reduction based on similarity of test cases", In: Proc. of 7th Brazilian workshop on systematic and automated software testing—CBSoft, 2013.

[27] C. Fang, Z. Chen, K. Wu, and Z. Zhao, “Similarity-based test case prioritization using ordered sequences of program entities”, Software Quality Journal, Vol.22, No.2, pp.335-361, 2014.

[28] S. Yoo, M. Harman, P. Tonella, and A. Susi, “Clustering test cases to achieve effective and scalable prioritization incorporating expert knowledge”, In: Proc. of the 18th international symposium on Software testing and analysis, pp. 201-212. ACM, 2009

[29] A. P. Mathur, “Foundations of Software Testing”, Pearson Education, 1st Edition, 2008.

[30] Rapps S, Weyuker EJ (1985) Selecting software test data using data flow information. IEEE Trans Softw Eng 4:367–375.

PUBLICATION DETAILS

[1] P. Ammu & M. Bharathi and Dr. C .P Indumathi, “Generating effective Test suite size using Similarity Based Greedy Approach”, published the paper in the proceedings of “UGC-SAP National Conference on Pattern Recognition, Informatics and Medical Engineering (PRIME-2019)”, paper ID: PRIME 19030 p.no:16, Periyar University ,Salem 21st -22nd March 2019.