IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 2, Ver. VI (Mar-Apr. 2014), PP 22-30
www.iosrjournals.org

Improving Fault Detection Capability Using Coverage Based Analysis

Sanyogita Chaturvedi[1], A.Kulothungan[2]
[1](*Department Of Computer Science & Engineering,SRM University,India)*
[2(]*Department Of Computer Science & Engineering(Asst. Professor),SRM University,India)*

Abstract: *A test suite for a system consists of several test cases and as the number of test cases is more, execution of all of these will take much time. If the number of test cases are reduced the execution time could be reduced. Moreover, if the execution of these is done in an ordered fashion ,it is observed to give an increased rate of fault detection. This can further be beneficial by providing feedback to system developers , improving fault fixing activity and thus ,software delivery. The reduction of test cases can be done in several ways. But the scenario is different when functional dependencies exists between some test cases, that is, one test case is needed to be executed before another. In this paper, two test case reduction techniques will be presented which will be followed by using the dependency information from a test suite. The first reduction technique considers all of the test cases while the second one is applied on the rejected suite obtained after implementing the first technique. This work is actually based on the hypothesis that lesser number of test cases will reduce the execution time, provided that the reduced test suite contains cases which collectively will cover the testing of every statement in the source code. Existing dependencies between different tests represent the interaction in the system under test, and hence, execution of complex instructions earlier is supposed to increase the fault detection rate, compared to arbitrary or untreated test orderings. Thus, the dependency information could further be used for prioritization of the test cases.*

Keywords: *test case reduction, functional dependency, regression testing.*

I. Introduction

The development of software systems involves a series of production activities where opportunities for injection of human fallibilities are enormous. Errors might occur at the very inception of the process where the objectives may be erroneously specified, as well as later design and development stages. Because of the very known human inability to perform and communicate with perfection, software development is accompanied by a quality assurance activity. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation[3]. Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to "demolish" the software that has been built.

Testing is a process of executing a program with the intent of finding an error. A good test case is one that has a high probability of finding an undiscovered error and a successful test is one that uncovers an as-yet-undiscovered error[3]. If testing is conducted successfully, it will succeed in covering errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to the specification, that behavioral and performance requirements appear to have been met. In addition, data collected when testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing does not show the absence of errors and defects, its objective is only that software errors and defects are present. It is very important to keep this statement in mind as testing is being conducted.

Each time a new module is added in the existing software as part of integration testing, the software changes. New data flow paths get established, new I/O may occur, and new control logic is invoked. These newly introduced modules may cause problems with functions that previously worked flawlessly. Regression testing is the re-execution of some subset of tests that have previously been conducted to ensure that changes have not propagated an unintended side effect. In a broader context, successful tests result in the discovery of errors, and errors must be corrected. Whenever software is corrected, it means that some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that the newly introduced changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

• A representative sample of tests that will exercise all software functions.

• Additional tests that focus on software functions that are likely to be affected by the change.

• Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Software testing is the major process in software development life cycle .So more importance is given to the testing process .Due to the large size of code and complexity testing becomes a most tedious job. Whenever the modification is done in the software the testing process is repeated so we need to rerun all the test cases so that the time taken for the testing process is increased .Instead we can give priority to the test cases and we can select the test cases depends upon the priority given to the test case. The priority is given depends upon the code coverage. There is many existing algorithm for performing test case prioritization and selection.

Retesting or Regression is the process of Repeating the Testing job with the same set of test cases and additional test cases that are generated after modification is done. Regression testing is used during the development phase and maintenance phase of a software product to assist software-testing activities and guarantee that quality is achieved through various releases of the software product . The Regression testing also called as Retesting which once again test the software with the existing test cases and the new test cases that is generated depending up on the change that is performed on the software .So that we are repeating the testing process by rerunning all the test cases. This will take extra time for testing .

Our major goal is to minimize the test suite so that the time required for the testing will be reduced.To perform Test Suite minimization, the amount of test cases is to be reduced by giving priority to the test cases that is present
in the test suite. Certain Testing tools are available for performing regression testing. There are many algorithms used for test case prioritization such as greedy, additional greedy, hill climbing ,Additional 2 Greedy Algorithm ,Heuristic Algorithm and Genetic Algorithm .Each algorithm is having their own functionality. The algorithms will differ in terms of the performance.

To perform testing, a very important metric to be used is cyclomatic complexity. It is based on the number of decisions in a program. Its importance lies in the fact that it provides an indication of the amount of testing necessary to practically avoid defects. In other words, areas of codes identified as more complex are candidates for reviews and additional dynamic tests. While there are many ways to calculate cyclomatic complexity, the easiest way is to sum the number of binary decision statements(e.g. if, while, for, etc.) and add 1 to it.

Mathematically, the cyclomatic complexity of a structured program is defined with reference to the control flow of the program, a directed graph containing the basic blocks of the program,with an edge between two basic blocks if control may pass from the first to second. The complexity M is then defined as

$$M=E-N+P$$

where,

E=number of edges of the graph
N=number of nodes of the graph
P=number of connected compon-
ents(exit nodes)

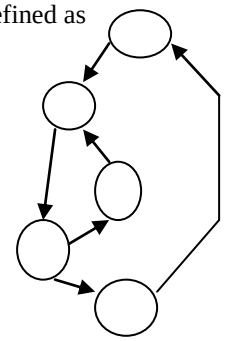

Fig 1 Control Flow Graph

A strongly connected control flow graph is shown. This graph has 6 edges, 5 nodes and 1 connected component, so the cyclomatic complexity of the program is

$$6-5+1=2$$

II. Literature Survey

The size of a test-suite has a direct impact on the effort and the costs of software testing. Common test-suite reduction techniques select subsets of test-suites that achieve given test requirements. Unfortunately, not only the test-suite size but also the fault detection ability is reduced as a consequence.

Software testing is a process which is known to consume a large part of the effort and resources involved in software development. Especially during regression testing, when software is re-tested after some modifications, the size of the test-suite has a large impact on the total costs. Therefore, the idea of test-suite reduction(also referred to as test-suite minimization) is to find a minimal subset of the test-suite that is sufficient to achieve the given test requirements. Various heuristics have been proposed to provide a minimal subset of the test-suite. These techniques can reduce the number of test-cases in a test- suite significantly.

CBR is one of the most popular and actively researched areas in the past used for test case reduction[2]. The researches done previously[4], [8], [16], [17] show that CBR has identical problems as same as software testing topic. In software testing field, particularly during regression testing activities, mainly researched issues are: (a) too many redundancy test cases after reduction process (b) a decrease of test cases' ability to reveal faults and (c) uncontrollable grow of test cases. Meanwhile, the key research issues in CBR field are: (a) there are too many redundancy cases in the CBR system (b) a size of CBR system is continuously growing all the time and (c) existing CBR deletion algorithms take longer time to remove all redundancy cases in the CBR system. These issues in CBR field can be elaborated as follows: Fundamentally, there are four steps in the CBR system, which are: retrieve, reuse, revise and retain. These steps can lead to a serious problem of growing cases in the system which is not desirable. However, the study shows that there are many proposed techniques in order to control the number of cases in the CBR system, such as add algorithms, deletion algorithms and maintenance approaches. CBR have been investigated by CBR researchers in order to ensure that only a very small amount of efficient cases are stored in the case base while others are ignored.

Definition 1: Barry [4] defined the CBR as follows:

"Case-Based Reasoning is one of the Artificial Intelligence-based algorithms, which solve the problems by searching through the case storage for the most similar cases. CBR has to store their solved cases back to their memory or storage in order to learn from their experience."

Another approach used for test case reduction proposes a novel technique where test- cases created with model-checker based techniques are transformed such that redundancy within the test-suite is avoided, and the overall size is reduced. As test-cases are not simply discarded, the impact on the fault sensitivity is minimal.

While performing redundancy based test case reduction it is said that a test-case contains redundancy if part of the test-case does not contribute to the fault detection ability[12]. The aim is to identify such redundancy, and also describe possibilities to reduce it. Intuitively, identical test-cases are redundant. For any two given test-cases t1, t2 such that t1 = t2, any fault that can be detected by t1 is also identifiable by t2 and vice versa, assuming the test-case execution framework assures identical preconditions for both tests. Similarly, the achieved coverage is identical for both t1 and t2 for any coverage criterion. Clearly, a test-suite does not need both t1 and t2 and thus one of the two could be ignored. The same consideration applies to two test-cases t1 and t2, where t1 is a prefix of t2. t1 is subsumed by t2, therefore any fault that can be detected by t1 is also detected by t2 (but not vice versa). In this case, t1 is redundant and is not needed in any test-suite that contains t2. In model-based testing it is common practice to discard subsumed and identical test- cases at test-case generation time [13].

III. Problem Description

3.1 Existing Problem

The reduction procedure used earlier are very simple and involve only a single technique through which reduction is done. However, following this method the resulting set of reduced test cases does not cover all the paths of the dependency graph for the given code.

Also, previous work on test case prioritization demonstrates that prioritization techniques are effective for improving rate of fault detection. However, these approaches do not consider test suites that contain functional dependencies between tests. Functional dependencies are the interactions and relationships among system functionality determining their run sequence. As test cases mirror this functionality, they also inherit these dependencies; therefore, executing some test cases requires executing other test cases first.

Such techniques which do not consider the existence of functional dependencies among test cases uses any order like breadth first search or random order for the prioritization of test cases.

Using these techniques the achieved fault detection rate is not high and so these prove to be inefficient as the main goal of a testing case prioritization is to find maximum number of faults in least possible time.

3.2 Proposed Solution

In this paper, we propose a new technique for test case reduction in which firstly we get a reduced set of test cases by a simple coverage based reduction method and further we take the rejected suite of test cases

from the coverage based method and perform a partial coverage based reduction. This finally gives a reduced set of test cases which collectively cover every path of the dependency graph and are then prioritized using an optimization technique.

Functional test case prioritization is based on the inherent structure of dependencies between tests, which we call dependency structure prioritization. Given that these dependencies reflect the dependencies of the system itself, it is proposed that ordering test executions based on the complexity of interactions between tests can increase the fault detection rate as compared with arbitrary test orderings. Our hypothesis is that, as a result, faults will be revealed earlier because scenarios containing more relationships are more complex and more fault prone.

IV. Proposed Technique

In the proposed technique, the entire processing is done following the design diagram given below. The diagram shows a two step reduction technique which firstly takes as input, the source code and all the test cases generated for the function and then performs a coverage based test case reduction. Another step is to perform partial coverage based reduction which gives the final reduced test suite.

After the completion of both the processes, lastly the dependency structure of the function is taken which is used to find the dependency between test cases of the reduced suite for each path in the dependency graph.

4.1 Proposed Methodology

In this paper, we will take a source code which is a small function performing few simple operations. Then the coverage area for the source code is found. The source code takes few inputs on which it works and different combinations of the inputs make several test cases i.e, test case generation process. Now, as the total number of test cases is large, we need to reduce them which is done using a simple coverage based reduction method. The process will give a reduced suite of test cases but these do not cover every path of the dependency graph of the source code.

Hence we consider the rejected test suite and again perform reduction using the partial coverage based technique. The partial coverage method basically considers every test case from the rejected suite of test cases with one input value missing from it and the statement covered is calculated. Then we finally get a reduced set of test cases. Further, the dependency structure for the function f1 is drawn and dependency between test cases is observed for all paths.

4.2 Design Diagram

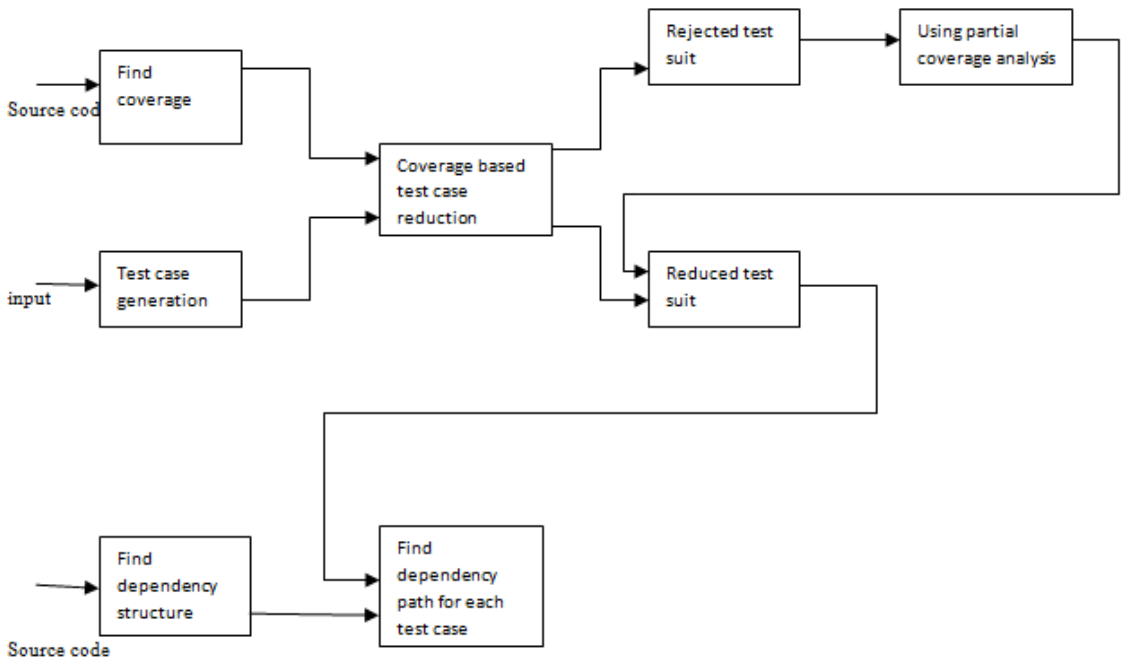

Fig 2 Design Diagram

Algorithm 4.1 Coverage Based Test Case Reduction:

```
1.  Reduced test suite, T=ϕ
2.  while(∃ Ti != ϕ)
3.  {for ti= 1 to n
      t=ϕ
4.   {for j=1 to n
5.     { if val[j]==1
6.         Ti= {t Ụ s[j] ; if ¬seen(Ti)
               { ϕ        ; otherwise
       } }
7.   No. of statements executed by Ti=#{Ti}
8.   max T= find max[#{Ti}]
9.   T=T Ụ max T
10.  if(Ti==max T || Ti Є T)
11.   Ti=seen(Ti)
12. Goto 2}

    where ti =test cases{1….n}
          j=statement no.
```

Algorithm 4.2 Partial Coverage Based Test Case Reduction:

```
1.  Reduced test suit, T=ϕ
2.  For i=1 to n
3.   { t= ϕ, #a[i][j]=0,sum[i]=0
4.    For j=1 to 3
5.    { for k= 1 to m
6.      { if val[k]==1
7.        { a[i][j]=t Ụ s[j]
8.            #a[i][j]= #a[i][j]+1
          } }
9.      sum[i]=sum[i]+a[i][j]
      } }
10. while(t<n)
11.  {¥ ¬seen sum[i],find max {sum[i]}
12.   T=T Ụ max{sum[i]}
13.    Max {sum[i]}=seen
14.     t++
      }
```

V. Case Study

In this paper, a small function is taken that performs simple operations. The source code needs three input values from the user. Here, it is explained how the proposed algorithm works on the source code taken. The source code is given below:

```
f1(A,B,C)
  {
    Int v=9; val[]=0;…….(1)
    val[1]=1;
    if(A<=V) ……..(2)
      val[2]=1;
      V=A+V;  ……(3)
      val[3]=1;
    else
      V=A-V;  ……(4)
      val[4]=1;
    if(B<=V) ……..(5)
      val[5]=1;
      B=B+V;  ……(6)
      val[6]=1;
      if(B>5)  …….(7)
```

```
  val[7]=1;
  printf("B");…….(8)
  val[8]=1;
if(C<V) ……...(9)
 val[9]=1;
 C=C+2; ……...(10)
 val[10]=1;
else
    C=C+V; ……(11)
    val[11]=1;
     if(C<8) …….(12)
      printf("C");
      val[12]=1;
```

In the source code, after every statement val[ ] is assigned as 1 which means that whenever a statement gets executed the value will be 1 and hence it could easily found out whether a test case executes a statement or not. The algorithms given uses this for its calculation of reduced suite.

5.1 Test Case Generation

Different inputs for the source code are taken which collectively form the test suite. The variables in the function f1 are A, B and C. These are provided with three different inputs and so

combination of all make 27 different test cases. Values taken for the variables are:
A={2,5,10}
B={6,3,2}
C={4,8,9}

Test case t1: A=2,B=6,C=4
t2 : A=2,B=6,C=8
and so on

5.2 Coverage Based Test Case Reduction

Taking the inputs as explained in previous section there are 27 different cases. Execution of these many test cases will take much time and so we need to reduce this number so that execution time could be reduced. But it has to be kept in mind that all the statements in the function are being executed by the reduced number of test cases.
The reduction is performed in several steps. Firstly, corresponding test suits for every statement is obtained. This test suit is actually the set of test cases executing the particular statement. Test suit for few statements are given below:

S1:{t1,t2,t3,t4,t5,t,6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16,t17,t18,t19,t20,t21,t22,t23,t24,t25,t26,t27}
S2:{t1,t2,t3,t4,t5,t,6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16,t17,t18}
S3:{t1,t2,t3,t4,t5,t,6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16,t17,t18}
S4:{ t18,t19,t20,t21,t22,t23,t24,t25,t26,t27}

Now, for every test case the list of statements it is executing is obtained. For ex:

t1: {s1,s2,s3,s5,s6,s7,s8,s9,s10}
t2: {s1,s2,s3,s5,s6,s7,s8,s9,s10}
t3: {s1,s2,s3,s5,s6,s7,s8,s9,s11,s12}

This gives the weight of the test cases i.e, the no. of statements each test case is executing.the test case executing maximum number of statements is included in the reduced test suit list.If more than one test case execute maximum no. of statements then one of those is taken and rest are ignored. Repeating the procedure , the reduced test suit is obtained which together execute all the statements of the function f1. The reduced suit obtained is:

T={t1,t3,t19}
All the remaining test cases are added in the rejected test suit {R}.

5.3 Partial Coverage Based Reduction

The reduced test suit obtained few no. of test cases but these are not able to cover all the paths of the dependency graph drawn for the function. To solve the issue more no. of test cases are generated using the rejected suit of test cases obtained from the previous section.This is done by taking the test cases from R and assigning one of the three variables a null value. When done for every variable,a single test case gives three new test cases with one variable missing in each case. This is shown below:

For t2:

t2,1: A=2,B=6,C=x
t2,2: A=2,B=x,C=8
t2,3: A=x,B=6,C=8

similarly for t4:

t4,1: A=2,B=3,C=x
t4,2: A=2,B=x,C=4
t4,3: A=x,B=3,C=4
and so on.

This way several new test cases are generated from the previously existing cases in the rejected suit. Now every test case newly obtained is calculated for the number of statements it is executing. For the subsets of t2 and t4 ,no. of statements being executed are:

t2,1:7 ; t2,2:5 ; t2,3:7
t4,1:7 ; t4,2:6 ; t4,3:8

Then these individual subsets are taken together and the no. of statements they are executing collectively is obtained by adding the corresponding no.
t2=(t2,1+t2,2+t2,3)
=7+5+7
=19

t4=(t4,1+t4,2+t4,3)
=7+6+8
=21

Shown above is the calculation only for two of the test cases from the rejected suit. Similarly all of the test cases are processed and later the test case with highest value is added to the reduced test suit. Again ignoring the previously selected test case ,case with highest value is added to the reduced suit.
Finally the reduced test suit becomes:

T={t1,t3,t4,t19,t27}

For the given source code,dependency structure is drawn which shows which statement within the function is dependent on any other statement.

5.5 Finding Dependent test cases for each path

In this section, the test cases are observed for having dependency ,following a particular path. This gives the idea for which test case is highly dependent on other test cases and also about which path within the dependency structure contains higher no. of dependencies.The purpose behind this is to perform proper ordering of the test cases for their execution. The dependency structure for the code is drawn.

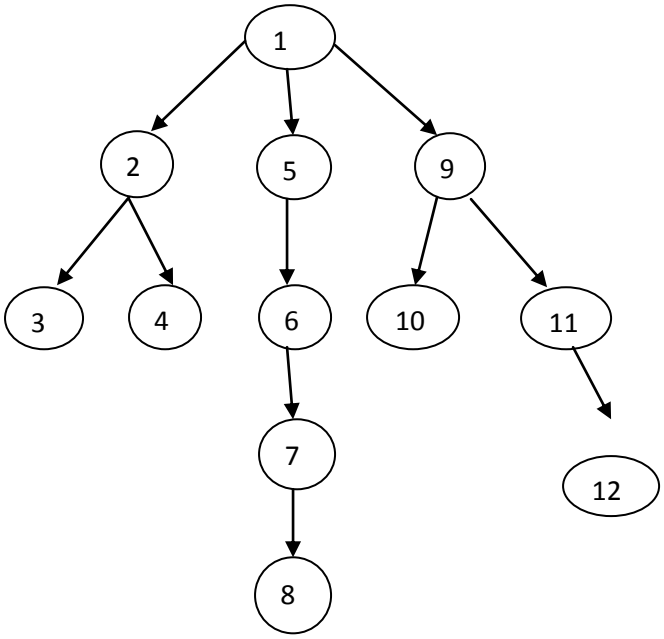

Fig 3 Dependency Graph

Path i: 1 → 2 → 3
Path ii: 1 → 2 → 4
Path iii: 1 → 2 → 5 → 6 → 7 → 8
Path iv: 1 → 9 → 10
Path v: 1 → 9 → 11 → 12

Path followed by test case t1 is Path i, iii & iv
Path followed by test case t3 is Path i, iii & v
Path followed by test case t4 is Path i, iii & v
Path followed by test case t19 is Path ii, iii & iv
Path followed by test case t27 is Path ii, iii & v

VI. Conclusion and Future Work

In this paper, a new technique for reduction of test cases in a test suite has been defined. The technique is basically a two step process where in first step all the existing number of test cases are considered and then a reduced test suite is formed containing few of the test cases initially taken while all the remaining test cases are grouped under the rejected suite. Further, in the second step, the rejected suite of cases is taken and a reduction procedure is applied which adds few more test cases in reduced suite formed in first step. The resulting test suite finally contains minimum number of test cases which are needed to be executed and collectively execute all of the statements in the source code. Also they cover all the paths of the dependency graph drawn for the source code. This implies that the source code, if tested using the cases obtained in reduced test suite, will give all the existing errors.

Further, we plan to use the dependency information extracted lastly which tells about the dependency existing between the test cases in a path for the prioritization of the test cases for their execution. The ordering of test cases if done using the dependency information is expected to give an increased rate of fault detection. The dependencies between tests represent the interaction in the system under test and executing complex interactions earlier is likely to increase the fault detection rate, compared to arbitrary test orderings [11],[18].

References

[1] J. Bach, "Useful Features of a Test Automation System (Part iii)," Testing Techniques Newsletter, Oct. 1996.
[2] Siripong Roongruangsuwan and Jirapun Daengdej, "Test Case Reduction Methods By Using CBR" Assumption University.
[3] Roger S. Pressman, "Softwae Engineering, A Practitioner's Approach",Fifth Edition.
[4] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", TRW Defense Systems Group, 1998.
[5] R.W. Floyd, "Algorithm 97: Shortest Path," Comm. ACM, vol. 5, no. 6, p. 345, June 1962.
[6] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin and Christie Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites", In *Proceedings of IEEE International Test Conference on Software Maintenance (ITCSM'98)*, Washington D.C., pp. 34-43, 1998.

[7] Heimdahl, M.P.E., Devaraj, G.: Test Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In: ASE, IEEE Computer Society (2004) 176–185.
[8] David C. Wilson. Ph.D. Thesis "A Case-Based Maintenance: The husbandry of experiences." Department of Computer Science, Indiana University, 2001.
[9] Gargantini, A., Heitmeyer, C.: Using Model Checking to Generate Tests From Requirements Specifications. In: ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume 1687., Springer (1999) 146–162
[10] Glenford J. Mayers (2004)," The Art Of Software Testing", John Wiley & Sons Publication.
[11] K.S. Lew, T.S. Dillon, and K.E. Forward, "Software Complexity and Its Impact on Software Reliability," IEEE Trans. Software Eng., vol. 14, no. 11, pp. 1645-1655, Nov. 1988.
[12] Gordon Fraser and Franz Wotaw, "Redundancy Based Test-Suite Reduction", Graz University of Technology.
[13] Ammann, P., Black, P.E., Majurski, W. Using Model Checking to Generate Tests from Specifications. In: ICFEM. (1998)
[14] Graves T, Harrold MJ, Kim JM, Porter A, Rothermel G. An empirical study of regression test selection techniques. Proceedings of the 20th International Conference on Software Engineering (ICSE 1998), IEEE Computer Society Press, 1998; 188–197.
[15] Tallam S, Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. SIGSOFT Software Engineering Notes 2006; 31(1):35–42.
[16] Jirapun Daengdej, Ph.D. Thesis, "Adaptable Case Base Reasoning Techniques for Dealing with Highly Noise Cases" The University of New England, Australia, 1998.
[17] Scott McMaster and Atif Memon, "Fault Detection. Probability Analysis for Coverage-Based Test Suite. Reduction", IEEE, 2007.
[18] C. Perrow, Normal Accidents: Living with High-Risk Technologies. Princeton Univ Press, 1999.

Sanyogita Chaturvedi-Received a B.tech(CSE) first class degree from Gautam Buddh Technical University in 2012. Currently she is persuing M.tech(CSE) from SRM University.

A.Kulothungan- Received a B.E degree and a M.E degree from Anna University.Currently he is an Assistant Professor in Computer Science department.His area of interests are software engineering, NLP and web technology