COMPONENTS IMPACT ANALYZER WITH GENETIC ALGORITHM

D. Jeyamala 1, K. Sabari Nathan 2, S. Balamurugan 3 and A. Jalila 4
Department of Computer Applications, Thiagarajar College of Engineering, India

E-mail: 1 djmcse@tce.edu, 2 sabarinathan4you@gmail.com, 3 balmsg4u@gmail.com and 4 mejalila@gmail.com

Abstract
High quality software can be obtained by means of rigorous testing of all the components of the software. This research work has proposed an automated software testing framework that performs a mutant based components impact analysis to identify the higher critical components from the Software Under Test (SUT). In this work, the mutants are automatically generated by injecting faults in the original program and they are used to identify the impact over the other components in the SUT. The generated mutants are executed using a suite of test cases to identify their impact over the other components of the system. Based on their impact level, the critical components are identified and then rigorously verified using the test cases generated using Genetic Algorithm (GA) based approach with branch coverage and mutation score based test adequacy criterion as the fitness functions. For unit testing, the branch coverage based test case adequacy criteria is used to test whether all the branches have been covered or not. In integration testing, the components are tested against the test cases generated using GA by means of identifying the execution trace of each method and each intermediate results is compared against the expected output stored in the repository. The testing tool named as “Impact Arbiter” developed as part of this work has carried out all these tasks in an automated way and has generated various graphs for the purpose of visualization.

Keywords:
Critical Components, Impact Analysis, Mutation Analysis, Genetic Algorithm (GA), Branch Coverage Value (BCV)

1. INTRODUCTION

Software testing is an important phase of Software Development Life Cycle (SDLC) [13]. Since exhaustive testing of software is not possible, but as per the Pareto’s principle [14], only 20% of the components have higher impact than the remaining components, and therefore we need to monitor and rigorously test only those 20% of the components during testing. This will not only improve the quality but also will reduce the testing cost and time. Hence there is a need to identify the critical components and apply rigorous testing on them prior to release. Hence, this approach has proposed a novel approach to identify the impact level of the 20% components which cause 80% of the problems after delivery.

The proposed approach applies a novel methodology namely mutant based component impact analysis. This is achieved by artificially injecting faults to the component and then identifying the impact level of the faulty component over the other components. As fault is an external, incorrect behaviour of a program that leads to incorrect result or failure, here faults are introduced by applying the Offutt mutation operators [15], [24]. It is said to be Arithmetic Operator Replacement (AOR), Relational Operator Replacement (JOR), Unary Operator Inclusion (UOI), Logical Connector Replacement (LCR), Absolute Keyword Inclusion (ABS) and it used to make change to the components automatically [5].

The fault identification is done by means of executing the mutants over test cases. The next task is to identify the execution trace to find their impact over the other components. Then, based on the outcome of the results, a complete impact analysis is performed by examining the impact level of each of the faulty versions of the component over the other components [1]. In this research work, the impact level is classified as catastrophic, critical, marginal and minor [1]. The higher impact may result from flawed procedures that cause catastrophic effects. Based on this mutation-based impact analysis, the overall impact level of each component is identified. The components which have higher impact are called as critical components

Once the critical components are identified, they are verified rigorously using the test cases generated using GA. The algorithm begins with a random set of test cases called as individuals. The test cases are chosen based on the mutation score which have higher fault adequacy generated based on random test cases. The test cases generated using Genetic Algorithm is used in unit testing and integration testing to test each of the components in the system. In unit testing, the code instrumentation is done for all the methods of the SUT to monitor the execution of each branch and related information. The generated test cases using GA are executed against each component to monitor the execution of each branch of the method. If branch coverage reaches above 98%, the unit testing process is said to be completed. Otherwise, the remaining set of test cases will be generated based on the coverage value using GA until the fitness value reaches above 98%. In the case of integration testing, the components are tested with the test cases generated using GA by means of identifying execution trace of each method and its intermediate result is compared against expected output stored in the repository.

The various reports and graphs which are generated as part of this research work show a clear picture about the overall view of the SUT against impact analysis, test cases efficiency, cost of testing process and so on. As an outcome of this approach, the critical components are identified using and their impact level and are verified using Genetic algorithm based test case generation and optimization.

2. RELATED WORK

P. K. Suri et al. [9] have designed a simulator to identify critical components in a component based system (CBS). In their work, they have used Component Execution Graph (CEG) which is a network representation of the CBS. In this graph they have assigned a weight for each execution link which is actually the weight of the destination component. Weight ‘W’ of an execution path is the sum of all ‘Wi’s of execution links along that path. They have assumed that each execution path with maximum weight is called the “Critical Execution Path” and execution links falling along that path are all critical execution
links and all the components falling on this path are the critical components.

Zhou et al. [10] have analyzed Object-Oriented design metrics for predicting high and low severity faults. Their results are based on public domain NASA data set. In their study they stated that, design metrics such as CBO, WMC, RFC, and LCOM metrics were statistically significant to find fault-proneness of classes across fault severity and the prediction capabilities of these metrics depend on the severity of faults. Also, they insisted that, the design metrics are better predictors of low severity faults in classes than high severity faults.

Shatnawi et al. [11] have experimented the effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. In their study they have tested software metrics such CBO, CTA (Coupling through Abstract Data Type), CTM (Through Message Passing), RFC, WMC, DIT, NOC etc. They proved that software metrics are used to identify error prone classes even after the software release evolution process.

Ray et al. [12] has proposed an analytical method for reliability-based risk assessment of a software system at the architectural level, which is based on UML sequence diagram and state chart diagram. In their work they have considered risk associated with various states of a component, message criticality and business risk to identify high risk components.

Gosev et al. [13] have applied UML and the commercial modelling environment Rational Rose Real Time (RoseRT) to obtain UML model statistics. In their approach, for each component and connector in software architecture, a dynamic heuristic risk issue is obtained and severity is assessed supported risk analysis. Then a Markov model is constructed to obtain scenario’s risk factors. The risk factors of use case and the overall system risk factors are estimated using the scenarios risk factors.

Lanubile et al. [16] has proposed to identify the software complexity measure using the modeling techniques like principal component analysis, layered neural networks, discriminant analysis, logical classification models, logistic regression, and holographic networks.

Jacek et al. [17] proposed the approach that identifies the fault prone components based on the risk assessment of impact of such post-release change fixes. The present their experiences with CRANE: a failure prediction, change risk analysis and test prioritization system at Microsoft Corporation that leverages existing research for the development and maintenance of Windows Vista. They identify and evaluate the impact and risk of a change is to understand the exact extent of changes.

Ohlsson et al. [18] has proposed using design metrics to identify the fault prone components with emphasis on the use of appropriate statistical methods to support quality improvement of software and they had taken Ericson Telecom AB as case study.

Birt et al. [19] has proposed using Genetic algorithm to predict faulty classes and identify accuracy of fault proneness prediction using Object oriented metrics.

As part of our previous work we published the following paper:

Mala and Praba [15] has proposed a novel regression testing methodology to identify and verify critical components, it can be done by means of dependability metrics and internal complexity metrics to calculate criticality measure and test those components using regression testing.

D. Jeya Mala and K. Sabari Nathan [23] proposed mutation based component impact analysis to identify the critical components in the SUT.

D. Jeya Mala and S. Balamurugan [24] used sensitivity and severity metrics to calculate the criticality measure of the SUT and to identify the critical components.

D. Jeya Mala and A. Jalila used the Object Constraint Language (OCL) Formal Specification design metrics to calculate the complexity of each component and based on the outcome, the critical components are identified in the SUT [25], [26].

3. PROPOSED WORK

The Software under Test (SUT) refers to any Java oriented real time system. Once it is given us input, the next task is to extract all the components, classes and methods in it. Here we undertook various case studies to verify this proposed approach. The following steps are involved:

Step 1: Extraction of SUT
Step 2: Perform Mutation based Impact Analysis to identify Critical Components.
Step 3: Critical Component Verification using Genetic Algorithm based Test Case Generation and optimization
Step 4: Unit and integration testing using GA
Step 5: Experimentation results of proposed approach
Step 6: Comparative analysis with Random testing.

From the above steps are categorized into the following phases:

i. Identification of Critical Components using Mutation based Impact Analysis.

4. PHASE 1- IDENTIFICATION OF CRITICAL COMPONENTS

4.1 MUTANTS GENERATION

Here we generate mutants for each component in the SUT. The faults are injected using the set of five mutation operators (i.e., ABS, AOR, ROR, LCR, and UOI) which would be more effective as all the 22 mutation operators of Mothra [8], a mutation-testing tool.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute Value Insertion</td>
<td>x = 2<em>a; -&gt; x = 2</em>abs(a);</td>
</tr>
</tbody>
</table>

Table 1. Offutt mutation operators
These five mutation operators have proved as sufficient mutation operators to generate a set of first order mutants [4].

In this step, we create mutants for each method in a component. Operators in each method can be inserted or replaced by Offutt operators which are mentioned in Table.1. In Fig.1 the 'checkBal()' method is taken from the "CheckBalance" component and changed the AOR mutation operator to replace '+' to '-' and the mutant is stored in the mutants list.

4.2 MUTATION ANALYSIS

Testing contains three main phases: test case generation, test execution, and test evaluation. A test case has components that describe an input, action or event under which a tester will determine if a feature of a SUT is working correctly or not [14].

Test case generation is the process of generating a collection of test cases which are applied to the SUT [14]. Here we generate random test data for each parameter in a method. The procedure is repeated for all components in the SUT. Apply the test cases to the component as a unit to rigorously cover it by means of executing the original and the mutants. Based on these results, the mutation score (MS) is evaluated and is used to identify the test case adequacy. The mutation score (MS) always lies between 0 and 1. If MS (T) = 0, the test case cannot distinguish any mutants and test cases is not efficient [15]. If MS (T) = 1 then test case distinguishes all mutants except those equivalent mutants and the test case is adequate to be applied in impact analysis [1]. The score is calculated based on method-wise, components-wise and application-wise mutation score in the SUT [1].

The following example shows the sample code, test cases and the execution results obtained after test case execution have been shown in Fig.2.

![Mutant Class1](image1)

```java
public class mut49 extends Account {
    public int viewBalance(String ano, String atp)  {
        -------------- code --------------
        if (ano.length()>0 || atp.length()==0) {
            -------------- code --------------
            return 0;
        }
        else {
            -------------- code --------------
            while (r.next()) {
                b = r.getInt(1);
            }
        }
        -------------- code --------------
        return b;
    }
}
```

Mutant Class2

```java
public class mut49 extends Account {
    public int viewBalance(String ano, String atp)  {
        -------------- code --------------
        if (ano.length()>0 || atp.length()==0) {
            -------------- code --------------
            return 0;
        }
        else {
            -------------- code --------------
            while (r.next()) {
                b = r.getInt(1);
            }
        }
        -------------- code --------------
        return b;
    }
}
```

Fig.1. Mutant Generation

Test Case 1:

Mut49 - {"","cur"}
Status : Distinguished

Mut50 - {"","cur"}
Status : Distinguished

Method wise Mutation Score : 1.0
Component wise Mutation Score : 1.0

Test Case 2:

Mut 49 - {"a153","cur"}
Status : Distinguished

Mut 50 - {"a153","cur"}
Status : Live

Method wise Mutation Score : 0.5
Component wise Mutation Score : 0.5

Fig.2. Mutation Score Calculation

**Application-wise Mutation Score:**

\[
MS (T) = \frac{|D|}{(|D| + |L|)}
\]  

where,

MS (T) – Mutation Score for test case T
D - No of Distinguished Mutants i.e. the mutants killed by the Test case T
L - No of Live Mutants i.e. the Test case T which could not kill the mutants

Method-wise Mutation Score:
\[ M_{S_m} (T) = |D_m| / (|D_m| + |L_m|) \]

where,
- \( M_{S_m} (T) \) - Mutation Score for Method-\( m \) against test case \( T \)
- \( D_m \) - No of distinguished mutated methods for Method-\( m \), i.e., the mutant for Method-\( m \) is killed by test case \( T \)
- \( L_m \) - No of live mutated methods for Method-\( m \), i.e., the mutant for Method-\( m \) not killed by Test case \( T \)

Component-wise Mutation Score:
\[ M_{S_c} (T) = |D_c| / (|D_c| + |L_c|) \]

where,
- \( M_{S_c} (T) \) - Mutation Score for Component-\( c \) against Test case \( T \)
- \( D_c \) - No of distinguished mutated components for Component-\( c \), i.e., the mutant for Component-\( c \) is killed by Test case \( T \)
- \( L_c \) - No of live mutated components for Component-\( c \), i.e., the mutant for Method-\( m \) not killed by Test case \( T \)

4.3 IMPACT ANALYSIS

The Coupling or dependency is the degree to which each component depends on other modules [14]. Cohesion means the degree to which the elements of a component work together to produce a single functionality. In this approach, all the connected components for each component are extracted based on cohesion and coupling measure among components [1]. The interconnection may be in the form of inheritance or message passing over other components [14]. The efficient test cases are chosen based on mutation score and execute over mutated components to identify the impact level of each component based on the execution trace, to know how far it affected over connected components in a system [1]. The mutants are generated based on Step 3.3.1. The impact is categorized as catastrophic, critical major and minor [1]. This categorization is explained below:

4.3.1 Catastrophic:

The outcome of a mutated method throws an exception or decides the control flow of the client which calls this method is called catastrophic [1].

4.3.2 Critical:

For example, the outcome of the mutated ‘validateAcc ()’ method of “Customer” class is used in the decision statement of the “CheckBalance” component’s validate () method. Here, the result of the ‘validate’ method will generate erroneous results due to the fault in the ‘validateAcc ()’ method and the entire application will collapse. Similarly, if the ‘validateAcc ()’ method throws an exception because of mutation means, the entire application will be terminated. Hence, the components that have this type of impact level are called as higher critical components.

4.3.3 Marginal:

For example, the outcome of mutated “balanceEnquiry()” method of “Transaction” class is in computational statement; hence the result of the method will generate erroneous results. So the impact level of component is critical.

Fig.3. Mutated method used in decision statement – Catastrophic

Fig.4. Mutated method used in the computational statement – Critical
public class Account {
    public String accountDetails(String a) {
        if (r.next()) {
            SI = p*n*r*100; // Mutated Statement
        }
        return SI.toString();
    }
}

public class Statements {
    public void display(String ano) {
        Account report = new Account();
        for (int i = 0; i < n; i++) {
            report.accountDetails(i);
        }
    }
}

Fig. 5. Mutated method used inside a looping statement

The mutated “accountDetails()” method of the “Account” class is called in looping statement; the result of the function will generate erroneous report. So the impact level of the component is marginal.

4.3.4 Minor:

If the outcome of a method is called only a few times in other components, then that component is said to have minor impact level.

public class CheckBalance{
    public int CheckBal(String ano, String at, int ba) {
        int j = Validate(accno, atp);
        if (j > 1) {
            // Mutated Statement
        }
        return value;
    }
}

Fig. 6. Mutated method simply invoked a component

For example, the mutated “checkBal()” method of the “CheckBalance” class is called in “Deposit” class, and the result of the function would give erroneous answers but impact level is very low compared to other categories. So the impact level of the component is minor.

5. PHASE 2 - CRITICAL COMPONENTS VERIFICATION

5.1 GA BASED TEST CASE GENERATION AND OPTIMIZATION

This novel approach uses a genetic algorithm for an optimization heuristic that minimizes the normal processes, such as selection and mutation in natural advancement. It begins with a random set of individuals (chromosomes) and through a crossover and mutation operations, gradually evolves the population toward an optimal solution based on the mutation score and the branch coverage value.

5.1.1 Pseudo Code:

i. Instrumentation of code is done for all components in the SUT
ii. Randomly initialize population (t)
iii. Determine fitness of population based on the mutation score (t)
iv. Repeat
   a. Select parents from population (t)
   b. Perform crossover and mutation on parents creating population (t+1)
   c. Determine fitness of population based on the mutation score and the branch coverage value (t+1)
Until best individual is good enough.

5.1.2 Crossover:

Choose a random point on the two individuals which act as parents. Split individuals at this crossover point. Create individuals by means of exchanging test cases which act as parents.

Example:

Test Case1: {1000,"xxxx","FD", 1599}
Test Case2: {1010,"yyyy","CUR", 10009}
Crossover point: 2
Test Case1: {1000,"xxxx","CUR", 10009}
Test Case2: {1010,"yyyy","FD", 1599}

5.1.3 Mutation:

Choose any test data in a test case which act as a parent. Mutate the test data using any one of bitwise or arithmetic operators for numeric data and random word for string data and so on.

Example:

Test Case: {1000,"xxxx","FD", 30}
Mutation point: 4
Test Case: {1000,"xxxx","FD", 31}

5.1.4 Fitness function:

The Generated test cases using crossover and mutation are executed against both original and mutant. Finally, the results are assessed and test adequacy is calculated using the following formulae.

Mutation Score

The Mutation Score is calculated using the above formulae (1, 2, 3).
where,

\[ MS(T) = (|DM| + |LM|) * 100 \]  

(4)

\[ Branch \ Coverage \]  

BCV (T) = (|BC| + |TB|) * 100  

(5)

\[ MS(T) - \text{Mutation Score for Test case T} \]  

DM - Distinguished Mutant - if results are different  

LM - Live Mutant - if the results are same i.e. Test case cannot reveal the error

\[ Branch \ Coverage \]  

BCV (T) = (|BC| + |TB|) * 100

\[ \text{MS(T)} - \text{Mutation Score for Test case T} \]  

DM - Distinguished Mutant - if results are different  

LM - Live Mutant - if the results are same i.e. Test case cannot reveal the error

5.2 UNIT TESTING USING GA

In unit testing, the code instrumentation is done for all methods of the SUT to monitor the execution of each branch for branch coverage based fitness function for individuals. The initial population is a random set of individuals. From them, the best two parent test cases are chosen based on the higher fitness values. The selected individuals are passed to crossover and mutation to generate further test cases and then evaluated based on the branch coverage value and the mutation score. Based on the fitness value, the remaining set of test cases will be generated and the test case generation will be continued until the fitness value reach above 98%.

5.3 INTEGRATION TESTING USING GA

In Integration testing, the code instrumentation is done for all methods of the SUT and components are tested against the efficient test cases generated using GA by means of identifying the execution trace of each method and each intermediate result is compared against the expected output in the repository and it shows the pair-wise class name; also, it shows the status of intermediate results.

Table.2. Integration Testing Results for sample component

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method Name</th>
<th>Pair wise Class</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckBalance</td>
<td>CheckBal()</td>
<td>Month</td>
<td>Pass</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>CheckBal()</td>
<td>Customer</td>
<td>Pass</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>CheckBal()</td>
<td>Statements</td>
<td>Pass</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>CheckBal()</td>
<td>Year</td>
<td>Pass</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>CheckBal()</td>
<td>CheckBalance</td>
<td>Fail</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>CheckBal()</td>
<td>CheckBalance</td>
<td>Pass</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>Validate()</td>
<td>CheckBalance</td>
<td>Pass</td>
</tr>
</tbody>
</table>

Example:

Parent Method is starting (“Class name”)
Child Method1 is starting (“Class name”)
Grand Child Method1 is starting (“Class name”)
Grand Child Method1 is ending (“Class name”, “return value”)
Child Method1 is ending (“Class name”, “return value”)
Child Method2 is starting (“Class name”)
Child Method2 is ending (“Class name”, “return value”)
Parent Method is ending (“Class name”, “return value”)

6. PERFORMANCE EVALUATION

6.1 EXPERIMENTATION SETUP

The proposed approach has been tested against four Application Software [AS] and 3 Real-Time System [RT], listed in Table.3.

Table.3. Case Studies Under Taken

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Test Problems</th>
<th>Test Object No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Blood Bank</td>
<td>AS1</td>
</tr>
<tr>
<td>2</td>
<td>Banking Application</td>
<td>AS2</td>
</tr>
<tr>
<td>3</td>
<td>Library</td>
<td>AS3</td>
</tr>
<tr>
<td>4</td>
<td>Hospital</td>
<td>AS4</td>
</tr>
<tr>
<td>5</td>
<td>Apache.ant.1.2 [20]</td>
<td>RT1</td>
</tr>
<tr>
<td>6</td>
<td>JWWalk [21]</td>
<td>RT2</td>
</tr>
<tr>
<td>7</td>
<td>Flaka [22]</td>
<td>RT3</td>
</tr>
</tbody>
</table>

Fig.7. Branch coverage based test adequacy assessment
6.2 SAMPLE CASE STUDY 1 – REAL TIME SYSTEM

The Table 4 shows impact analysis for real time system oriented sample case study, Apache Ant 1.2 [20]. Based on the impact analysis we have extracted critical components which are listed in Table 5.

Table 4. Impact Analysis for Apache-Ant 1.2

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Affect</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>junit3.AntUnitTestCase</td>
<td>junit4.AntUnitSuiteRunner</td>
<td>Minor</td>
</tr>
<tr>
<td>junit3.AntUnitTestSuite</td>
<td>junit3.AntUnitTestCase</td>
<td>Minor</td>
</tr>
<tr>
<td>junit3.AntUnitTestSuite</td>
<td>junit4.AntUnitSuiteRunner</td>
<td>Minor</td>
</tr>
<tr>
<td>AntUnitScriptRunner</td>
<td>AntUnit</td>
<td>Catastrophic</td>
</tr>
<tr>
<td>AntUnitScriptRunner</td>
<td>junit3.AntUnitSuite</td>
<td>Marginal</td>
</tr>
<tr>
<td>LogContent</td>
<td>LogContent</td>
<td>Critical</td>
</tr>
<tr>
<td>LogContains</td>
<td>LogContains</td>
<td>Critical</td>
</tr>
<tr>
<td>ResourceExists</td>
<td>Exception*</td>
<td>Catastrophic</td>
</tr>
<tr>
<td>LogCapturer</td>
<td>LogContent</td>
<td>Marginal</td>
</tr>
<tr>
<td>LogCapturer</td>
<td>LogContains</td>
<td>Marginal</td>
</tr>
<tr>
<td>junit3.MultiProjectDemuxOutputStream</td>
<td>junit3.AntUnitSuite</td>
<td>Minor</td>
</tr>
<tr>
<td>AssertTask</td>
<td>Exception*</td>
<td>Catastrophic</td>
</tr>
<tr>
<td>AntUnit</td>
<td>Exception*</td>
<td>Catastrophic</td>
</tr>
</tbody>
</table>

*Because of mutation it throws an Exception and it is also called as Show-Stoppers*

Table 5. Critical Components for Apache Ant

<table>
<thead>
<tr>
<th>Critical Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AntUnitScriptRunner</td>
</tr>
<tr>
<td>ResourceExists</td>
</tr>
<tr>
<td>AssertTask</td>
</tr>
<tr>
<td>AntUnit</td>
</tr>
</tbody>
</table>

6.3 SAMPLE CASE STUDY 2 – APPLICATION SOFTWARE

The Table 6 shows impact analysis for application oriented sample case study, Banking system. Based on the impact analysis we have extracted critical components which are listed in Table 7.

Table 6. Impact Analysis for Banking System

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Class Affect</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>Admin</td>
<td>Critical</td>
</tr>
<tr>
<td>Admin</td>
<td>User</td>
<td>Minor</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>Deposit</td>
<td>Minor</td>
</tr>
<tr>
<td>CheckBalance</td>
<td>Withdraw</td>
<td>Minor</td>
</tr>
</tbody>
</table>

CheckBalance  | CurrentAcc  | Minor    |
CheckBalance  | Transaction | Catastrophic |
CheckBalance  | Account     | Minor    |
CheckBalance  | OnlineServices | Minor  |
ChequeBook    | User        | Critical  |
ChequeBook    | Exception   | Catastrophic |
CurrentAcc    | Account     | Minor    |
Customer      | CheckBalance | Catastrophic |
Deposit       | FundTransfer | Catastrophic |
FixedAcc      | Account     | Minor    |
FundTransfer  | Transaction | Minor    |
Month         | Customer    | Minor    |
OnlineServices| ProfileUpdate | Catastrophic |
ProfileUpdate | User        | Minor    |
ProfileUpdate | ChequeBook  | Minor    |
RecurringAcc  | Account     | Minor    |
SavingAcc     | Account     | Minor    |
Statements    | Year        | Marginal  |
Statements    | .Month      | Minor    |
Transaction   | Account     | Minor    |
User          | Admin       | Critical  |
User          | Transaction | Minor    |
Withdraw      | FundTransfer | Catastrophic |
Year          | CheckBalance | Minor    |

Based on the outcome of impact analysis, the higher critical component is extracted and listed in Table 7.

Table 7. Critical Components for Banking Application

<table>
<thead>
<tr>
<th>Critical Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit</td>
</tr>
<tr>
<td>CheckBalance</td>
</tr>
<tr>
<td>OnlineServices</td>
</tr>
<tr>
<td>Customer</td>
</tr>
<tr>
<td>Deposit</td>
</tr>
<tr>
<td>Withdraw</td>
</tr>
<tr>
<td>ChequeBook</td>
</tr>
</tbody>
</table>

As per the Infosys White Paper “Realizing Efficiency and Effectiveness in software through a Comprehensive metrics model” [14], the following graphs have been generated.

6.3.1 Connected Components Graph:

It shows the cohesion and coupling measure of all components and it also shows its connected components which have been extracted as in Step 4.3. It has been shown in Fig.10.
6.3.2 Show-Stoppers’ Trend Graph:
A Show-Stopper is an exception thrown while during execution, which generally has higher impact thereby making software dysfunctional. Tracking show-stoppers is very important and hence we have identified and representation in the form of the graph as shown in Fig.9.

6.3.3 Impact Level Graph:
By implementing Step 4.3 the components’ impact value has been obtained. Based on the impact level, the higher impact level components are extracted and represented in the graph. This graph has showed the complete set of critical components and their impact value. The graph depicts the critical components and their overall impact level in the SUT as shown in Fig.10.

6.3.4 Cost of Testing Component wise Graph:
The objective of this graph is to identify software components having intensive test effort areas and identify the areas that need improvement actions. The graph in Fig.11 is generated using the following formulae.

\[
\text{Cost of each component} = \frac{\text{No of test cases} \times \text{Test case cost} \times \text{impact value}}{\text{Component of Cost}}
\]

6.3.5 Defects Detected % trends Graph:
Defect Removal Efficiency (DRE) of each component is used to determine the effectiveness of Genetic Algorithm Test case defect removal efforts. It serves as an oblique indicator of the quality of the product. It measure the no of defects reported during mutation and how it reveal by the Genetic Algorithm Test case during each cycle. The higher percentage of DRE is the most positive impact on component quality. This is because it represents the ability of Genetic Algorithm to kill mutants. The
highest possible value of DRE is “1” or “100%”. The following formulae are used to calculate DRE. It is shown in Fig.12.

\[ \text{DRE} = \left( \frac{|\text{TM}|}{|\text{TM}| / |\text{AM}|} \right) \times 100 \]  

(7)

where,

\( \text{TM} \) = Total number of mutant components
\( \text{AM} \) = No of Mutants revealed by GA in each cycle

6.3.6 Overall Test Coverage Graph:

The graph depicts the amount of component/test cases actually covered branches successfully via the total number of components/test cases of the SUT. It is an important factor to measure the effectiveness of the testing process. The Fig.13 shows the effectiveness of the testing process of the SUT.

6.3.7 Component-wise Test Coverage Graph:

The graph depicts the amount of test cases actually covered branches successfully via the total number of test cases of each component. It is an important factor to measure the effectiveness of the testing process for each component. The Fig.14 shows the effectiveness of the testing process of components.

### Table 8. Random based Optimization

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Test Problems</th>
<th>BCV (%)</th>
<th>MS</th>
<th>Execution Time in seconds</th>
<th>Total no. of Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AS1</td>
<td>56%</td>
<td>0.65</td>
<td>16.374</td>
<td>413</td>
</tr>
<tr>
<td>2</td>
<td>AS2</td>
<td>75%</td>
<td>0.83</td>
<td>1417.258</td>
<td>1855</td>
</tr>
<tr>
<td>3</td>
<td>AS3</td>
<td>69%</td>
<td>0.9</td>
<td>75.99</td>
<td>1543</td>
</tr>
<tr>
<td>4</td>
<td>AS4</td>
<td>80%</td>
<td>0.78</td>
<td>589.22</td>
<td>1759</td>
</tr>
<tr>
<td>5</td>
<td>RT1</td>
<td>60%</td>
<td>0.8</td>
<td>2913.521</td>
<td>1799</td>
</tr>
<tr>
<td>6</td>
<td>RT2</td>
<td>65%</td>
<td>0.77</td>
<td>5957.45</td>
<td>3765</td>
</tr>
<tr>
<td>7</td>
<td>RT3</td>
<td>55%</td>
<td>0.65</td>
<td>5793.56</td>
<td>3753</td>
</tr>
</tbody>
</table>

*The case studies contains equivalent mutants.

### Table 9. GA Based Optimization

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Test Problems</th>
<th>BCV (%)</th>
<th>MS</th>
<th>Execution Time in seconds</th>
<th>Total no. of Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AS1</td>
<td>99%</td>
<td>0.98</td>
<td>11.875</td>
<td>254</td>
</tr>
<tr>
<td>2</td>
<td>AS2</td>
<td>98%</td>
<td>0.99</td>
<td>1167.288</td>
<td>1125</td>
</tr>
<tr>
<td>3</td>
<td>AS3</td>
<td>98%</td>
<td>0.98</td>
<td>63.28</td>
<td>1353</td>
</tr>
<tr>
<td>4</td>
<td>AS4</td>
<td>99%</td>
<td>0.95 *</td>
<td>365.47</td>
<td>736</td>
</tr>
<tr>
<td>5</td>
<td>RT1</td>
<td>97%*</td>
<td>0.97 *</td>
<td>2354.843</td>
<td>1509</td>
</tr>
<tr>
<td>6</td>
<td>RT2</td>
<td>96%</td>
<td>0.96</td>
<td>3453.78</td>
<td>2974</td>
</tr>
<tr>
<td>7</td>
<td>RT3</td>
<td>97%*</td>
<td>0.97 *</td>
<td>4693.756</td>
<td>3225</td>
</tr>
</tbody>
</table>

*The case studies contains infeasible branches.
It indicates that the performance of GA is superior and takes only less time for to completely test the critical components. It takes only 70% of test cases when compared to test cases generated using Random. Execution time is also reduced. The coverage value and mutation score of GA are improved than coverage value and mutation score of random based test cases.

The Fig.15 shows that more number of test cases generated using Random are failed in the case studies.

The Fig.16 shows only few test cases of GA are failed in the case studies, because those branches are infeasible branches.

The Fig.17 shows that the non-linear optimization of Random achieves same mutation score for more number of times and it cannot reach 0.98.

The Fig.18 shows the local optimization of Genetic Algorithm and reaches the maximum mutation score.

7. CONCLUSION

The proposed approach that can automatically generate mutants to identify the critical components based on the impact level of the components. The test cases are then executed to both original and mutants, based on the results, mutation score is calculated. The mutation score is test adequacy criteria to identify the impact levels of all the components over its connected components and based on the impact level the critical components are identified and it is verified using GA by means of unit testing and integration testing. The optimized test cases
are stored in the repository for future use. During the enhancement of software, the tester can use these test cases to test the software up to the underlined-level. Finally, based on the results, the efficiency of GA and Random are compared and represented in the form of graphs to make an assessment about the proposed approach. In future we will apply other test case optimization techniques like Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO) and so on, for a comparative study.

ACKNOWLEDGEMENT

This research paper is part of the UGC Major Research Project supported by University Grants Commission, New Delhi, India.

REFERENCES