CRITICAL COMPONENTS IDENTIFICATION USING MUTATION BASED COMPONENTS IMPACT ANALYSIS

D. JEYAMALA
Department of Computer Applications, Thiagarajar College of Engineering, Tamilnadu – 625 015, djmcse@tce.edu

K. SABARI NATHAN
Department of Computer Applications, Thiagarajar College of Engineering, Tamilnadu – 625 015, sabarinathan4you@gmail.com

A. JALILA
Department of Computer Applications, Thiagarajar College of Engineering, Tamilnadu – 625 015, mejalila@gmail.com

S. BALAMURUGAN
Department of Computer Applications, Thiagarajar College of Engineering, Tamilnadu – 625 015, balamsg4u@gmail.com

Follow this and additional works at: https://www.interscience.in/ijcsi

Part of the Computer Engineering Commons, Information Security Commons, and the Systems and Communications Commons

Recommended Citation
Available at: https://www.interscience.in/ijcsi/vol4/iss1/11

This Article is brought to you for free and open access by Interscience Research Network. It has been accepted for inclusion in International Journal of Computer Science and Informatics by an authorized editor of Interscience Research Network. For more information, please contact sritampatnaik@gmail.com.
CRITICAL COMPONENTS IDENTIFICATION USING MUTATION BASED COMPONENTS IMPACT ANALYSIS

D. JEYAMALA¹, K. SABARI NATHAN², A. JALILA³, S. BALAMURUGAN⁴

¹,²,³,⁴Department of Computer Applications, Thiagarajar College of Engineering, Madurai, Tamil Nadu, India
E-mail: djmcscl@tce.edu¹, sabarinathan4you@gmail.com², mejalila@gmail.com³, balamsg4u@gmail.com⁴

Abstract—High quality software can be obtained by means of resolving the complexity of the software. According to Pareto principle, 20% of components lead to 80% of the problems [1]. So, we need to identify those 20% of the components during testing. Therefore, this research work suggested an automated software testing framework to identify critical components using mutant based dynamic impact analysis for Software under Test (SUT). Mutants are automatically generated by injecting faults in the components using Offutt mutation operators and they are utilised to identify their impact level over other components of the system. The generated mutants and original program are executed using the suite of test cases, based on the conclusion of both the results, the mutation score is assessed and furthermore it is utilised as the test case adequacy criterion to recognize the impact level of it over the other components of a system. The outcome of this innovative approach is a testing tool entitled as JImpact Analyzer that automates the entire task and has generates miscellaneous graphs for visualization purpose.

Software Testing

Keywords- Software under Test (SUT), Critical Components, Impact Analysis, Mutants, Mutation Analysis

1. INTRODUCTION

Software testing is an important phase of the software development process and ensures that defects are identified as early as possible [2]. This paper proposes a novel idea to identify the impact level of such 20% components that originate 80% of the difficulty after delivery. The proposed approach provides a novel methodology namely mutant based dynamic impact analysis. This is achieved by artificially injecting faults to the components and then recognizing the impact level of each of the faulty component over the other components. As fault is an incorrect behavior of a program that directs the incorrect result or malfunction, here faults are introduced by applying the Offutt mutation operators[3] AOR, ROR, UOI, LCR, ABS, JTD, JTI operator by means of injecting changes to the components in the SUT.

The critical component Identification is done by executing the test cases over mutants and finding the execution trace to seek out their impact level over the other components. Then, on the basis of the outcome of the results, mutation score is calculated and is utilized as test case adequacy criterion and an entire impact analysis is performed by examining the impact level of each of the defective versions of the component over the residual components. The Impact level is categorized as catastrophic, critical, marginal and minor. The catastrophic effects are due to the results of flawed procedures. Based on the mutation analysis, overall impact level of each component is analyzed. The component that has higher impact is termed as critical components. As a part of this research work, various graphs and PDF reports have been created that shows a clear picture about the overall outlook of the SUT against impact analysis, test cases efficiency defect distribution and so on. As a result of this approach, a complete list of critical components is identified using mutation based dynamic impact analysis.

0 II. CRITICAL COMPONENT IDENTIFICATION

1. Related Work:

P.K Suri and Kumar [9] have conceived a simulator to recognize 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 allotted a weight for each execution connection which is actually the weight of the destination component. Weight „W” of an execution link is the sum of all „wi”s of execution links along that link. They have presumed that each execution route with greatest heaviness is called the “Critical Execution Path” and execution links dropping 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 analysed Object-Oriented design metrics for predicting high and low severity obvious errors. Their outcome is based on public domain NASA data set. In their study they asserted 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 proposition capabilities of these metrics depend on the severity of faults. Furthermore, they insisted
that, the design metrics are better predictors of reduced severity faults in classes than high severity faults.

Shatnawi et al. [11] have experimented the effectiveness of software metrics in recognizing 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 verified that software metrics are utilised to identify error prone classes even after the software release evolution process.

Ray and Mohapatra [12] have suggested an analytical procedure for reliability-based risk assessment of a software system at the architectural level which is founded on UML sequence diagram and state chart diagram. In their work they have advised risk affiliated with diverse states of a component, message criticality and business risk to recognize high risk components.

Goseva-Popstojanova et al. [13] have applied UML and 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 factor is recognized and severity is considered based on risk analysis. Then a Markov model is assembled to get scenarios risk factors. The risk factors of use case and overall system risk factor are approximated utilising the scenarios risk factors.

2. Proposed Work: Mutant Based Impact Analysis

2.1 Proposed Algorithm

Step 1: Read SUT

Step 2: Extract components from it.

Step 3: Generate test cases to execute against the SUT and store the results.

Step 4: Perform Impact Analysis using the following steps.

a) Mutant generation for each method in the components of the SUT.

b) Mutant execution done by means of a suite of test cases.

c) Extract connected components list.

d) Impact analysis of each component over the other components.

Step 5: Graph generation

a) Impact level graph

b) Connected components graph

c) Priority distribution for defects graph

d) Show Stoppers graph

Step 6: Report generation

a) Connected components report

b) Impact Analysis report

2.2 Software Under Test (SUT) Analysis

Choose any Java based real time system and extract all the class components, inner classes and methods in it. Here a Banking Application is taken as a case study to verify the proposed approach.

2.3 Program Execution

Testing is a significant stage of the Software Development Life Cycle (SDLC). A test case is a sequence of steps to check the correct behaviour or functionality and characteristic of a SUT is working correctly or not [6]. Program execution contains three main phases: Test case generation, Test case execution and Test evaluation. Test case generation is the method of evolving a suite of test cases which are directed to Software Under Test (SUT). Here we generate test data for all parameters in a method and execute them against components in the SUT and Mutation Score (MS) is considered for test evaluation. The procedure is repeated for all components in the software. Based on the mutation score based test case adequacy criterion, the efficiency of test case is identified. The mutation score is calculated based on the method-wise, components-wise and application-wise for the SUT. In this part, we tend to execute all parts of the SUT and appropriate test data are generated for each of its parameters in a method.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>-32666,-9566</td>
</tr>
<tr>
<td>short</td>
<td>2000,2000</td>
</tr>
<tr>
<td>long</td>
<td>49999999.56666</td>
</tr>
<tr>
<td>float</td>
<td>4562.55,45689</td>
</tr>
<tr>
<td>double</td>
<td>-0.86888.566,789,623</td>
</tr>
<tr>
<td>boolean</td>
<td>True,False</td>
</tr>
<tr>
<td>String</td>
<td>&quot;abedghijklmnopqrstuvwxyz&quot;,null,&quot;&quot;</td>
</tr>
<tr>
<td>Char</td>
<td>&quot;a&quot;,&quot;z&quot;,&quot;z&quot;,&quot;z&quot;,&quot;z&quot;,</td>
</tr>
<tr>
<td>Class&lt;-&gt;</td>
<td>new String(), new Integer()</td>
</tr>
<tr>
<td>Object</td>
<td>new String(&quot;abc&quot;), new Double(10.5)</td>
</tr>
<tr>
<td>byte</td>
<td>010101</td>
</tr>
</tbody>
</table>

| ClassName | new ClassName(10) |

<table>
<thead>
<tr>
<th>ClassName</th>
<th>MethodName</th>
<th>ParameterType</th>
<th>ParameterValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm.Transaction</td>
<td>display</td>
<td>String</td>
<td>nd</td>
</tr>
<tr>
<td>ibm.Transaction</td>
<td>validate</td>
<td>String</td>
<td>Admin</td>
</tr>
<tr>
<td>ibm.Transaction</td>
<td>validate</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ibm.User</td>
<td>Login</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ibm.User</td>
<td>Login</td>
<td>String</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ClassName</th>
<th>MethodName</th>
<th>ParameterType</th>
<th>ParameterValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm.User</td>
<td>Validate</td>
<td>String</td>
<td>nd</td>
</tr>
<tr>
<td>ibm.User</td>
<td>Validate</td>
<td>String</td>
<td>nd</td>
</tr>
<tr>
<td>ibm.Transaction</td>
<td>WithBalance</td>
<td>String</td>
<td>Admin</td>
</tr>
<tr>
<td>ibm.Transaction</td>
<td>WithBalance</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ibm.Transaction</td>
<td>WithBalance</td>
<td>int</td>
<td>2108326480</td>
</tr>
<tr>
<td>ibm.Withdraw</td>
<td>WithBalance</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ibm.Withdraw</td>
<td>Check</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ibm.Withdraw</td>
<td>Check</td>
<td>String</td>
<td>null</td>
</tr>
<tr>
<td>ibm.Withdraw</td>
<td>Check</td>
<td>int</td>
<td>-1429683559</td>
</tr>
</tbody>
</table>
Critical Components Identification using Mutation based Components Impact Analysis

2.4.2 Mutants Generation

Generate mutants for each method of a component using the set of mutation operators prescribed by Offutt [7]. If we use more number of mutation operators, it may lead to generating too many mutants. It may take exhaustive time or more memory space. Therefore, we use only a subset of mutation operators that can achieve approximately the same effect than using all operators. Offutt [2], [7] found that five mutation operators (i.e., ABS, AOR, ROR, LCR, and UOI) are roughly as productive as all the 22 mutation operators of Mothra [8], a mutation-testing tool. These five mutation operators are denoted as adequate mutation operators. The Component that does not have any of these overhead operators in the components means it can be injected fault utilising the Offutt class level mutation operators (i.e., JTD, JTI).

<table>
<thead>
<tr>
<th>Table: Test Case Execution for Case Study</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ClassName</strong></td>
</tr>
<tr>
<td>ibsm.Transaction</td>
</tr>
<tr>
<td>ibsm.Transaction</td>
</tr>
<tr>
<td>ibsm.User</td>
</tr>
<tr>
<td>ibsm.User</td>
</tr>
<tr>
<td>ibsm.Withdraw</td>
</tr>
<tr>
<td>ibsm.Withdraw</td>
</tr>
</tbody>
</table>

Fig-1 shows the sample code and sample test cases for evaluating it.

```java
public class Transaction {
    public int validate(String a, String atp) {
        if (a.length() == 0 || atp.length() <= 0) {
            return 0;
        }
        User us = new User();
        Validate(a, atp);
        int n; a;
        at = atp;
        CheckBalance cb = new CheckBalance();
        int i = cb, Validate(a, n, at);
        if (i != 0) {
            FundTransfer fund = new FundTransfer();
            fund.FundTransfer(a, atp, n, atp, i);
        }
        return i;
    }
    ---- code -----
}
```

Test Case for Validate :

T1("111", "Ed")
T2("112", "admin")

2.4 Mutation Based Impact Analysis

2.4.1 Pseudo code

1. Extract component Ci from the SU model where i = 1 to n.
2. Generate mutants Mi to Mi for Ci. Where n is total number of mutants.
3. Execute Test Cases TCi over Mi to Mn where j = 1 to n.
4. Examine the impact of Mi to Mn on execution of the test cases based on the categories II-Catastrophic, II-Critical, III-Marginal, and I-Minor.
5. Based on the categories, final set of critical components are identified.

2.4.2 Mutants Generation

Generate mutants for each method of a component using the set of mutation operators prescribed by Offutt [7]. If we use more number of mutation operators, it may lead to generating too many mutants. It may take exhaustive time or more memory space. Therefore, we use only a subset of mutation operators that can achieve approximately the same effect than using all operators. Offutt [2], [7] found that five mutation operators (i.e., ABS, AOR, ROR, LCR, and UOI) are roughly as productive as all the 22 mutation operators of Mothra [8], a mutation-testing tool. These five mutation operators are denoted as adequate mutation operators. The Component that does not have any of these overhead operators in the components means it can be injected fault utilising the Offutt class level mutation operators (i.e., JTD, JTI).

<table>
<thead>
<tr>
<th>Table: Offutt mutation operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abbreviation</strong></td>
</tr>
<tr>
<td>ABS</td>
</tr>
<tr>
<td>AOR</td>
</tr>
<tr>
<td>LCR</td>
</tr>
<tr>
<td>ROR</td>
</tr>
<tr>
<td>UOI</td>
</tr>
<tr>
<td>JTD</td>
</tr>
<tr>
<td>JTI</td>
</tr>
</tbody>
</table>

In this phase, we create mutants for each method in a component. The faults are introduced in each method by artificially changing the operator using Offutt’s mutation operators which are mentioned in Table 1.

```java
public class Transaction {
    public int validate(String a, String atp) {
        if (a.length() <= 0 || atp.length() <= 0) {
            return 0;
        }
        User us = new User();
        Validate(a, atp, n, a);
        int n; a;
        at = atp;
        CheckBalance cb = new CheckBalance();
        int i = cb, Validate(a, n, at);
        if (i != 0) {
            FundTransfer fund = new FundTransfer();
            fund.FundTransfer(a, atp, n, atp, i);
        }
        return i;
    }
    ---- code -----
}
```

Fig 2: Sample Mutant Generation
In Fig-2 depicts sample mutant generation for case study. Here the upper half of fig 2 shows „checkBal()” method of “CheckBalance” component is altered by the ROR operator to interchange of „>=” to „<=" and lower half of fig 2 shows „validate()” method of “Transaction” component is altered by means of inserting „this” keyword to the statement and store both mutants in the mutants list.

2.4.3 Generate Connected Components List

Cohesion and Coupling are a significant factor to measure how intra connection within the component and inter connection among components is made in SUT [1]. Cohesion means that the degree to which the elements of a component work along to produce a single functionality. Coupling or dependency is that the degree to which each component depends over other modules.

In this approach we extracted all the connected components for every component in SUT based on the cohesion and coupling measure. This coupling and cohesion are derived in term of inheritance or message passing over other components.

2.4.4 Mutants Execution

As per Step 2.3 the test cases are generated and applied to the component as a unit to rigorously test it by means of executing both the original and mutants and based on the outcomes, the mutation score is calculated and is used to calculate the test case adequacy. The Mutation Score (MS) always lies between 0 and 1 [2]. If MS (T) = 0, the test case cannot differentiate any mutants and it is not adequate to be applied for the impact analysis.

Method-wise Mutation score:

\[
MS_m(T) = \frac{|D_m|}{|D_m|+|L_m|} \quad (1)
\]

\( D_m \) - 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.)
Critical Components Identification using Mutation based Components Impact Analysis

Component-wise Mutation score:

\[ MS_{c}(T) = \frac{|D|}{|D| + |L|} \]  \hspace{1cm} (2)

- \( MS_{c}(T) \) - Mutation Score for Component(c) against test case (T).

\( D \) - No of distinguished mutated components for Component(c) (i.e. the mutant for Component(c) is not killed by test case(T)).

\( L \) - No of live mutated components for Component(c) (i.e. the mutant for Component(c) is not killed by test case (T)).

Application wise Mutation Score:

\[ MS(T) = \frac{|D|}{|D| + |L|} \]  \hspace{1cm} (3)

- \( MS(T) \) – Mutation Score against 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.

The following fig 4 shows sample mutated component for “CurrentAccount” class and how component-wise and method-wise mutation score is calculated using the formula (1) (2).

The Method-wise Mutation Score is calculated for Case Study using the formula (1)

```
public class mut33 extends Account {
    String an, at, sacn;
    int b;
    public int ViewBalance(String ano, String atpn) {
        if (ano.length() <= 0 &
        try {
            if (ano.length() <= 0 &
            public class mut34 extends Account {
    String an, at, sacn;
    int b;
    public int ViewBalance(String ano, String atpn) {
        if (ano.length() <= 0 || atp.length() <= 0) {
            return 0;
        }
    }
```

TABLE-VI Component-wise Mutat ion Score for Case Study

<table>
<thead>
<tr>
<th>Class name</th>
<th>Killed</th>
<th>Live</th>
<th>MutScore</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibsm.Account</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.Admin</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.CheckBalance</td>
<td>3</td>
<td>1</td>
<td>0.75*</td>
</tr>
<tr>
<td>ibsm.ChequeBook</td>
<td>4</td>
<td>2</td>
<td>0.666667*</td>
</tr>
<tr>
<td>ibsm.CurrentAcc</td>
<td>1</td>
<td>1</td>
<td>0.5*</td>
</tr>
<tr>
<td>ibsm.Customer</td>
<td>4</td>
<td>2</td>
<td>0.666667*</td>
</tr>
<tr>
<td>ibsm.Deposit</td>
<td>2</td>
<td>2</td>
<td>0.5*</td>
</tr>
<tr>
<td>ibsm.FixedAcc</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.FundTransfer</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.Month</td>
<td>2</td>
<td>2</td>
<td>0.5*</td>
</tr>
<tr>
<td>ibsm.OnlineServices</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.ProfileUpdate</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.RecurringAcc</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.SavingAcc</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsm.Statements</td>
<td>1</td>
<td>1</td>
<td>0.5*</td>
</tr>
<tr>
<td>ibsm.Transaction</td>
<td>3</td>
<td>1</td>
<td>0.75*</td>
</tr>
<tr>
<td>ibsm.User</td>
<td>2</td>
<td>2</td>
<td>0.5*</td>
</tr>
<tr>
<td>ibsm.Year</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The Method-wise Mutation Score is calculated for Case Study using the formula (1)
2.4.5 Impact Analysis

As per Step 2.3 the test cases are extracted with the most effective method-wise mutation score and are executed over the mutated components (i.e. which is generated based on Step 2.4.1) to find out the impact level of each component based on the outcomes of mutants and furthermore the execution trace is used to identify how far it influenced different parts during a system. The impact is categorized as follows

i) Catastrophic

The higher impact component or critical component will throw an exception or decide the control flow of the client.

### TABLE-VII Method-wise Mutation Score for Case Study

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Method Name</th>
<th>Killed</th>
<th>Live</th>
<th>Mut Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibsn.Account</td>
<td>Validate</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Account</td>
<td>DeleteAcc</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Admin</td>
<td>crate</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn/Admin</td>
<td>DeleteSubAc</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn/Admin</td>
<td>CreateAcct</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.CheckBalance</td>
<td>Validate</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.CheckBalance</td>
<td>CheckBal</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.CheckBook</td>
<td>GetDetails</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.CheckBook</td>
<td>ChkBrSt</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ibsn.CheckBook</td>
<td>ChkBr</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ibsn.CurrentAcc</td>
<td>ViewBalance</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Customer</td>
<td>validate</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Customer</td>
<td>GetDetails</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Customer</td>
<td>ViewBalanc</td>
<td>e 1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.Deposit</td>
<td>Check</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.Deposit</td>
<td>DebitBalanc</td>
<td>e</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.FixedAcc</td>
<td>ViewBalance</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.FundTransfer</td>
<td>FundTrans</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Month</td>
<td>display</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Month</td>
<td>validat     e</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ibsn.OnlineServices</td>
<td>Validate</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.ProfileUpdate</td>
<td>UpDetails</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.ProfileUpdate</td>
<td>GetDetails</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.RecuringAcc</td>
<td>ViewBalance</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.SavingAcc</td>
<td>ViewBalance</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Statements</td>
<td>Validate</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Transaction</td>
<td>display</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Transaction</td>
<td>validate</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.User</td>
<td>login</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.User</td>
<td>validate</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.Withdraw</td>
<td>Check</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>ibsn.Withdraw</td>
<td>WithBalanc</td>
<td>e</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ibsn.Year</td>
<td>validate</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

*The components which have Mutation Score < 1 contains equivalent mutants*

The mutated method “validate( )” of “CheckBalance” class is used in the decision statement of “Transaction” class. Here the outcome of the function is generating erroneous results due to the fault in “validate( )” and the entire software will collapse.

```java
public class CheckBalance {
    public int validate(String ano, String at) {
        // Mutated Statement
        if (ano != null) {
            return 0;
        }
        return 1;
    }
}
```

![Fig 5: Mutated method is in decision statement](image)

### Fig 5: Mutated method in decision statement

The mutated method “validate( )” of “CheckBalance” class is used in the decision statement of “Transaction” class. Here the outcome of the function is generating erroneous results due to the fault in “validate( )” and the entire software will collapse.

```java
public class Transaction {
    public void display(String ano) {
        // Mutated Statement
        cust.viewBalance(ano);
    }
}
```

![Fig 6: Mutated method Contains infinite loop](image)

### Fig 6: Mutated method Contains infinite loop
The mutated method “display()” of “Transaction” class throws an exception in “Account” class because of the infinite loop and the entire application will be terminated because of this mutation. Hence these components have higher impact level, and are identified as critical components.

ii) Critical
The outcome of a method is in computational statement. The mutated method of “getDetails()” of “ChequeBook” class is used in computational statement of “User” class and therefore the results of the function generate inaccurate results because of mutation as the impact level of this component is termed as critical.

```java
public class ChequeBook extends OnlineServices {
    public int GetDetails(String an, String at) {
        if (an != null || at.equals("")) //Mutated Statement
            return 0;
        else
            if (rs.next()) {
                return bal;
            }
    }
}
```

```java
public class User {
    public String Login(String uid, String pswd) {
        ChequeBook chk_bal = new ChequeBook();
        float bal = chk_bal.getDetails(uid, pswd);
    }
}
```

Fig 7: Mutated method is in Computational statement

iii) Marginal:
A method is called many times in other components. The mutated method of “viewBalance()” of “Customer” class is called in a looping statement of “Transaction” class the result of the function will generate inaccurate report. So the impact level of the component is marginal.

```java
public class Customer {
    public int viewBalance(String acctype, String atypeno) {
        if (acc.type.length() > 0 || atypeno.length() <= 0) {
            Customer cust = new Customer();
            for (int i = 0; i < n; i++)
                cust.ViewBalance(ano);
        }
    }
}
```

Fig 8: Mutated method is in Computational statement

iv) Minor
A method is called only a few times in other components. The mutated method “display()” of “Month” class is called in “Customer” class, the result of the function would give inaccurate results but impact level is very low compared to other categories. So the impact level of the component is minor.

```java
public class Month extends Statements {
    public void display(String ano) {
        if (ano.length() > 0) {
            while (rs.next())
                jTable1.setValueAt(r.getString(1), i, i);
        }
    }
}
```
Impact Analysis for Case Study

In this phase the components are extracted from case study and analyse the impact level of every mutated component over residual components is identified and listed in Table VIII.

2.5 Graph Generation

The subsequent graphs are generated as per Infosys White Paper “Realizing Efficiency and Effectiveness in software through a Comprehensive metrics model” [14].

2.5.1 Connected Components Graph

It displays the cohesion and coupling measure of all components and it furthermore displays its connected components that are extracted as in Step 2.4.3. It has been shown in fig 10.

2.5.2 Show-Stopper’s Trend Graph

A Show-Stopper is an exception thrown while throughout execution, which usually has higher impact which makes software dysfunctional. Extracting show-stoppers are very significant and thus we have recognized and representation in the pattern of the graph is shown in fig 11.

2.5.3 Impact Level Graph

By applying Step 2.4.5 the components’ impact level has been identified. This graph has shows the entire

Fig 9: Mutated method simply invoke by Component Impact Analysis for Case Study

Fig 10: Connected Components Graph

Fig 11: Show Stoppers Trend Graph
set of the components and their overall impact level in the SUT.

2.5.4 Priority Distribution of Defects Graph
The categories of impact are the priority distribution of defects outstanding against the SUT. During the end of the development phase this graph assists Final Gate Review teams to make an assessment of release readiness of the SUT. The graph shows in the fig 12.

2.6 Report Generation

2.6.1 Connected Components Report The tool generates PDF format of the report for connected components list. It is shown in the fig 14.

2.6.2 Impact Analysis Report
Component-wise failure distribution is done by means of mutant generation as in Step 2.4.2. Based on the execution of mutants, the impact level is analysed and therefore the report is generated based on it. This report displays the component wise failure distribution and their impact level over residual components. As considered in Step 2.4.5, the report in fig 14 displays the clear picture about all mutated components and their impact level over other components in the SUT.

III. CONCLUSIONS
We have developed an automated framework that can generate mutants automatically to identify the impact of the components based on the category such as catastrophic, critical, marginal, and minor. The test cases are executed to both original and mutants. Based on the execution, the results are compared and the impact levels of all the components are analyzed. As a result, a list of critical components has been recognized and listed. In future, rather than the test case with random test data we have the tendency to use test case optimization techniques like Genetic Algorithm, Hybrid Genetic Algorithm, Ant Colony Optimization and so on.

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

REFERENCES
Critical Components Identification using Mutation based Components Impact Analysis


V. APPENDIX

Fig 15: Read SUT

Fig 16: Connected Components extraction

Fig 17: Program Execution

Fig 18: Mutant List

Fig 19: Impact Analysis