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
JEYAMALA, D.; NATHAN, K. SABARI; JALILA, A.; and BALAMURUGAN, S. (2014) "CRITICAL COMPONENTS IDENTIFICATION USING MUTATION BASED COMPONENTS IMPACT ANALYSIS," International Journal of Computer Science and Informatics: Vol. 4 : Iss. 1 , Article 11.  
DOI: 10.47893/IJCSI.2014.1174
Available at: https://www.interscience.in/ijcsi/vol4/iss1/11

This Article is brought to you for free and open access by the Interscience Journals at 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: djmcse@tce.edu1, sabarinishan4you@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.

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” 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 met rics 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 el 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 demeanour 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.

| Data type | Examples          |
|-----------|-------------------|
| int       | -32666.9566      |
| short     | 2000,-2000       |
| long      | 49999999,-566666 |
| float     | 4562.55,45689    |
| double    | -986888.566789623 |
| boolean   | True,False       |
| String    | “abcdefhijklmnopqrstuvwxyz”,null,”” |
| Char      | “a”, “z”, “*”, ‘ ‘ |
| Class<?>  | new String(), new Integer() |
| Object    | new String(“abc”), new Double(10.5) |
| byte      | 010101            |
| ClassName | null, new ClassName(10) |

2.4 Test Case Generation using Random Data

| ClassName | MethodName | ParameterType | ParameterValue |
|-----------|------------|---------------|----------------|
| ibsm.Transaction | display | String | nd |
| ibsm.Transaction | validate | String | Admin |
| ibsm.Transaction | validate | String | svv |
| ibsm.User | Login | String | null |
| ibsm.User | Login | String | null |

| ClassName | MethodName | ParameterType | ParameterValue |
|-----------|------------|---------------|----------------|
| ibsm.User | Validate | String | nd |
| ibsm.User | Validate | String | nd |
| ibsm.Withdraw | WithBalance | String | Admin |
| ibsm.Withdraw | WithBalance | String | |
| ibsm.Withdraw | WithBalance | int | 2108326480 |
| ibsm.Withdraw | WithBalance | String | fd |
| ibsm.Withdraw | Check | String | |
| ibsm.Withdraw | Check | String | fd |
| ibsm.Withdraw | Check | int | -1425983559 |
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).

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.

| Abbreviation | Description | Example mutation |
|--------------|-------------|------------------|
| ABS          | Absolute Value Insertion | \( z = \text{abs}(b) \); |
| AOR          | Arithmetic Operator Replacement | \( z = a + b; \Rightarrow z = a - b; \) |
| LCR          | Logical Connector Replacement | \( z = \text{and}(a, b); \) |
| ROR          | Relational Operator Replacement | \( i \text{if}(a > b); \Rightarrow i \text{if}(a < b) \) |
| UOI          | Unary operator Inclusion | \( z = a + b; \) |
| JTD          | this keyword deletion | \( z = \text{this.x} + y; \Rightarrow z = \text{this.x} + y \) |
| JTI          | this keyword insertion | \( z = \text{this.x} + y; \Rightarrow z = \text{this.x} + y \) |

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.

![Fig 1: Sample Code and Test Cases](image1)

**TABLE-I**

| Class Name | Method Name | Return Type | Return Value | Execution Time |
|------------|-------------|-------------|--------------|----------------|
| ibsm.Transaction | display | void | 0 | 5 |
| ibsm.Transaction | validate | int | 1 | 53 |
| ibsm.User | Login | String | 111 | 32 |
| ibsm.User | Validate | String | | 10 |
| ibsm.Withdraw | WithBalance | int | 1 | 53 |
| ibsm.Withdraw | Check | int | 16 | |

**TABLE-II** Test Case Execution for Case Study

| Class Name | Method Name | Return Type | Return Value | Execution Time |
|------------|-------------|-------------|--------------|----------------|
| ibsm.Transaction | display | void | 0 | 5 |
| ibsm.Transaction | validate | int | 1 | 53 |
| ibsm.User | Login | String | 111 | 32 |
| ibsm.User | Validate | String | | 10 |
| ibsm.Withdraw | WithBalance | int | 1 | 53 |
| ibsm.Withdraw | Check | int | 16 | |

**TABLE-III**

| Class Name | Method Name | Return Type | Return Value | Execution Time |
|------------|-------------|-------------|--------------|----------------|
| ibsm.Transaction | display | void | 0 | 5 |
| ibsm.Transaction | validate | int | 1 | 53 |
| ibsm.User | Login | String | 111 | 32 |
| ibsm.User | Validate | String | | 10 |
| ibsm.Withdraw | WithBalance | int | 1 | 53 |
| ibsm.Withdraw | Check | int | 16 | |

![Fig 2: Sample Mutant Generation](image2)
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.

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

In fig 3 it has been shown how “Transaction” class invokes the function of “User” and “CheckBalance” class. Here interconnection is in form of object creation and message passing. The following Table-V shows connected components list based on method of interaction in SUT.

| Class Name       | Class Affected   | Method Name    |
|------------------|------------------|----------------|
| ibsm.Statement   | ibsm.Month       | Validate       |
| ibsm.Statement   | ibsm.Year        | Validate       |
| ibsm.Transaction | ibsm.Account     | display        |
| ibsm.User        | ibsm.Admin       | Validate       |
| ibsm.User        | ibsm.Transaction | Validate       |
| ibsm.Withdraw    | ibsm.FundTransfer| WithBalance    |
| ibsm.Withdraw    | ibsm.Withdraw    | validate       |
| ibsm.Year        | ibsm.CheckBalance| validate       |

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 efficient. If MS(T) = 1, the test case distinguishes all the mutants except the equivalent mutants and it’s adequate to be applied for the impact analysis.

Method-wise Mutation score:

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

\(MS_m(T)\) - 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:**

\[ MS_c(T) = \frac{|D_c|}{|D_c| + |L_c|} \]  

(2)

\[ MS_c(T) \] - 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 Component(c) is not killed by test case (T).

**Application wise Mutation Score:**

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

(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).

| Class name       | Killed | Live | MsScore |
|------------------|--------|------|---------|
| ibsm.Account     | 2      | 0    | 1       |
| ibsm.Admin       | 8      | 0    | 1       |
| ibsm.CheckBalance| 3      | 1    | 0.75    |
| ibsm.ChequeBook  | 4      | 2    | 0.66    |
| ibsm.CurrentAcc  | 1      | 1    | 0.5     |
| ibsm.Customer    | 4      | 2    | 0.66    |
| ibsm.Deposit     | 2      | 2    | 0.5     |
| ibsm.FixedAcc    | 1      | 0    | 1       |
| ibsm.FundTransfer| 2      | 0    | 1       |
| ibsm.Month       | 2      | 2    | 0.5     |
| ibsm.OnlineServices| 1     | 1    | 0.5     |
| ibsm.ProfileUpdate| 4     | 0    | 1       |
| ibsm.RecurringAcc| 2      | 0    | 1       |
| ibsm.SavingAcc   | 2      | 0    | 1       |
| ibsm.Statements  | 1      | 1    | 0.5     |
| ibsm.Transaction | 3      | 1    | 0.75    |
| ibsm.User        | 2      | 2    | 0.5     |
| ibsm.Withdraw    | 2      | 2    | 0.5     |
| ibsm.Year        | 1      | 1    | 0.5     |

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

**TABLE-VI Component-wise Mutation Score for Case Study**

Here the Component-wise Mutation Score is calculated for Case Study using the formula (2)
### TABLE VII: Method-wise Mutation Score for Case Study

| Class Name     | Method Name   | Killed | Live | Mut Score |
|----------------|---------------|--------|------|-----------|
| ibsn.Account   | validate      | 2      | 0    | 1         |
| ibsn.Admin     | DeleteAcc     | 2      | 0    | 1         |
| ibsn.Admin     | NewSubAcc     | 2      | 0    | 1         |
| ibsn.CheckBalance | validate     | 2      | 0    | 1         |
| ibsn.CheckBalance | checkBal    | 1      | 1    | 0.5      |
| ibsn.CheckBook | checkDetails  | 1      | 1    | 0.5      |
| ibsn.CheckBook | chkBkSt       | 0      | 2    | 0         |
| ibsn.CheckBook | chkBk         | 0      | 2    | 0         |
| ibsn.CurrentAcc | viewBalance   | 2      | 0    | 1         |
| ibsn.Customer  | validate      | 2      | 0    | 1         |
| ibsn.Customer  | checkDetails  | 2      | 0    | 1         |
| ibsn.Deposit   | check         | 1      | 1    | 0.5      |
| ibsn.Deposit   | debitBalance  | 1      | 1    | 0.5      |
| ibsn.FixedAcc  | viewBalance   | 2      | 0    | 1         |
| ibsn.FundTransfer | fundTrans   | 2      | 0    | 1         |
| ibsn.Month     | display       | 2      | 0    | 1         |
| ibsn.Month     | validate      | 0      | 2    | 0         |
| ibsn.OnlineService | validate   | 1      | 1    | 0.5      |
| ibsn.ProfileUpdate | upDetails  | 2      | 0    | 1         |
| ibsn.ProfileUpdate | getDetails  | 2      | 0    | 1         |
| ibsn.RecurringAcc | viewBalance | 2      | 0    | 1         |
| ibsn.SavingAcc | viewBalance   | 2      | 0    | 1         |
| ibsn.States    | validate      | 2      | 0    | 1         |
| ibsn.Transaction | display     | 2      | 0    | 1         |
| ibsn.Transaction | validate     | 1      | 1    | 0.5      |
| ibsn.User      | login         | 1      | 1    | 0.5      |
| ibsn.User      | validate      | 1      | 1    | 0.5      |
| ibsn.Withdraw  | check         | 1      | 1    | 0.5      |
| ibsn.Withdraw  | withdraw      | 1      | 1    | 0.5      |
| ibsn.Year      | validate      | 1      | 1    | 0.5      |

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

#### 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.

```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

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

```java
public class Account {
    public String validate(String an, String at) {
        Transaction ts = new Transaction();
        ts.display(at);
        return true;
    }
}
```

International Journal of Computer Science and Informatics, ISSN (PRINT): 2231 –5292, Volume-4, Issue-1
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;
            }
        }
    }
}
```

![Fig 7: Mutated method is in Computational statement](image)

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

```java
public class Month extends Statements {
    public void display(String ano) {
        if(ano.length()>0) //Mutated Statement
        {
        }
    }
}
```

![Fig 8: Mutated method is in Computational statement](image)

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 Customer {
    public int viewBalance(String acctype, String atypeno) {
        if (acctype.length() > 0 || atypeno.length() <= 0) {
            // Mutated Statement
            return 0;
        }
        return bal;
    }
}
```

![Fig 8: Mutated method is in Computational statement](image)
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.

| Mutant Id   | Class Affect   | Impact   |
|-------------|----------------|----------|
| ibsm.Account.mut1 | ibsm.Admin    | Critical |
| ibsm.Admin.mut5   | ibsm.User     | Minor    |
| ibsm.CheckBalance.mut11 | ibsm.Deposit | Minor    |
| ibsm.CheckBalance.mut11 | ibsm.Withdraw| Minor    |
| ibsm.CheckBalance.mut11 | ibsm.CurrentAge| Minor   |
| ibsm.CheckBalance.mut12 | ibsm.Transaction| Catastrophic |
| ibsm.CheckBalance.mut12 | ibsm.Account   | Minor    |
| ibsm.CheckBalance.mut12 | ibsm.OnlineServices| Minor   |
| ibsm.CheckBalance.mut15 | ibsm.User     | Critical |
| ibsm.CheckBook.mut18 | Exception    | Catastrophic |
| ibsm.CurrentAcc.mut21 | ibsm.Account  | Minor    |
| ibsm.Customer.mut24 | ibsm.CheckBalance | Catastrophic |
| ibsm.Deposit.mut29   | ibsm.FundTransfer| Catastrophic |
| ibsm.FundTransfer.mut33 | ibsm.Account| Minor    |
| ibsm.Month.mut37    | ibsm.Customer | Minor    |
| ibsm.OnlineServices.mut41 | ibsm.ProfileUpdate| Catastrophic |
| ibsm.ProfileUpdate.mut43 | ibsm.User   | Minor    |
| ibsm.ProfileUpdate.mut43 | ibsm.CheckBook| Minor    |
| ibsm.RecurringAcc.mut47 | ibsm.Account| Minor    |
| ibsm.SavingAcc.mut49 | ibsm.Account  | Minor    |
| ibsm.Statements.mut51 | ibsm.Year     | Minor    |
| ibsm.Statements.mut51 | ibsm.Month    | Minor    |
| ibsm.Transaction.mut53 | ibsm.Account| Minor    |
| ibsm.User.mut57    | ibsm.Admin    | Critical |
| ibsm.User.mut57    | ibsm.Transaction| Minor   |
| ibsm.Withdraw.mut62 | ibsm.FundTransfer| Catastrophic |
| ibsm.Year.mut65    | ibsm.CheckBalance| Minor   |

From Table VIII we have extracted critical components as it has more impact than all other components and it is listed in Table IX.

| Class Name        | Class Affect        | Impact       |
|-------------------|---------------------|--------------|
| ibsm.Deposit      | ibsm.FundTransfer   | Catastrophic |
| ibsm.CheckBalance | ibsm.Transaction    | Catastrophic |
| ibsm.OnlineServices| ibsm.ProfileUpdate| Catastrophic |
| ibsm.Customer     | ibsm.CheckBalance   | Catastrophic |
| ibsm.Deposit      | ibsm.FundTransfer   | Catastrophic |
| ibsm.Withdraw     | ibsm.FundTransfer   | Catastrophic |
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 fig14.

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
[1] Roger S. Pressman, “Software Engineering”, A Practitioner’s Approach 5th Edition, McGraw Hill, 1997.
[2] Aditya P.Mathur, “Software Testing”, Pearson Education, Asia
[3] A.J.Oftutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, “An experimental determination of sufficient mutation operators”, ACM Transactions on Software Engineering Methodology, 5(2):99-118, April 1996.
[4] MuJava homepage [Online]. Available: http://cs.gmu.edu/~offutt/mujava/
[5] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon, “MuJava: An Automated Class Mutation System”, Journal of Software Testing, Verification and Reliability, 15(2):97-133, June 2005.
Critical Components Identification using Mutation based Components Impact Analysis

[6] Srinivasan Desikan, Gopalaswamy Ramesh, “Software Testing Principles & Practices” PEARSON Education, 2006.

[7] A. Offutt, G. Rothermel, and C. Zapf, “An experimental evaluation of selective mutation,” in Proc. ICSE, 1993, pp. 100–107.

[8] A. R. A. DeMillo and R. I. Martin, “The Mothra software testing environment user’s manual”, Software Engineering Research Center, Tech.Rep., 1987.

[9] P K Suri, Kumar Sandeep, “Simulator for Identifying Critical Components for Testing in a Component Based Software System”, IJCSNS International Journal of Computer Science and Network Security, Vol.10, No.6, 2010; pp. 250-257.

[10] Zhou Yuming, Hareton Leung, “Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults”, IEEE Transactions on Software Engineering, Vol. 32, No. 10, 2006; pp.771-789.

[11] Shatnawi A Raed, Li Wei, “The Effectiveness of Software Metrics in Identifying Error-Prone Classes in Post-Release Software Evolution Process”, Journal of Systems and Software, Vol. 81, 2008; pp.1868–1882.

[12] Ray Mitribinda, Prasad Mohapatra Durga, “A novel methodology for software risk assessment at architectural level using UML diagrams”, SETLabs Briefings, Vol 9, No 4, 2011; pp.41-60.

[13] Goseva-Popstojanova Katerina, Hassan Ahmed, Guedem Ajith, Abdelmoez Walid, M.Nassar Diaa Eldin, Ammar Hany, Mili Ali, “Architectural Level Risk Analysis”, IEEE Transactions on Software Engineering, Vol. 29, No.10.2003; pp. 946 – 960.

V. APPENDIX