CRITICALITY ANALYZER AND TESTER – AN EFFECTIVE APPROACH FOR CRITICAL COMPONENT IDENTIFICATION & VERIFICATION USING ABC

D. Jeya Mala
Assistant Professor
Department of Computer Applications
Thiagarajar College of Engineering
Tamil Nadu – 625 706, India
djmcse@tce.edu

S. Balamurugan
PG Student
Department of Computer Applications
Thiagarajar College of Engineering
Tamil Nadu – 625 706, India
balamsg4u@gmail.com

K. Sabari Nathan
PG Student
Department of Computer Applications
Thiagarajar College of Engineering
Tamil Nadu – 625 706, India
sabarinathan4you@gmail.com

ABSTRACT
Now days, the Software industries has been evolved to satisfy the people's needs through developing the software. To satisfy their customers, the software industries need to track the critical components which may make serious impacts on the software. But tracking of the critical components is very important and most time consuming process. In this paper, we proposed a novel approach, Criticality Analysis which identifies the critical components of the software. The identified critical components are verified to ensure that are failure-free components using an intelligent search based optimization algorithm, Artificial Bee Colony.

Categories and Subject Descriptors
Category: D. SOFTWARE

D.2 SOFTWARE ENGINEERING

D.2.5 Testing and Debugging

Subject Descriptor: *Testing tool, Tracing.*

D.2.8 Metrics

Subject Descriptor: *Complexity measures*

General Terms
Algorithms, Measurement and Verification

Keywords
Artificial Bee Colony Algorithm, Branch Coverage, Criticality Analysis, Critical Components, Sensitivity Metrics, Severity Metrics, Test Optimization, Verification

1. INTRODUCTION
In Software industries, the customer being an important part of it and the industries doesn't operate without them. They decide the industries' growth or loss. Likewise, the Critical components are the important part of any software, and they decide the functionalities of the entire software. So, the critical components need some additional concentration while developing the software.

In our proposed approach, we introduce a novel methodology "Criticality Analysis" to identify the Critical components of the software. The criticality analysis involves two parts such as Sensitivity analysis and Severity analysis. The Sensitivity analysis is the process of extracting the coupling and cohesion metrics of the component. It identifies how a component will impact the other dependent components. Sensitivity analysis uses the existing metrics such as Fan-In, Fan-Out [1], and Information Flow [2]. In addition to that, we introduce the novel metrics as Weightage of Methods of a Component, Weakness of Methods of a Component and Ratio of Pure Inherited Methods for the Sensitivity analysis.

Severity analysis is the process of analyzing the impacts of failure. It identifies what type of failure that component might have on the software. The failure types are categorized as per the approach proposed by Vahid Garousi [3].

Artificial Bee Colony algorithm is an optimization which implied from the intelligent search behavior of honey bees to solve complex problems. Testing is most crucial and time consuming process in the Software development process. Many researches [7] had proven that this algorithm produces the efficient solution for testing. We incorporate this algorithm to optimize the testing process.

2. RELATED WORKS
Ebert Christof [4] has evaluated classification techniques such as Pareto classification, Classification Trees, Factor-Based Discriminant Analysis, Fuzzy Classification and Neural Networks for identifying critical components to predict faults based on code complexity metrics. His study showed that among those classification techniques, fuzzy classification provides the best result for critical component identification. Also, they insisted that, Pareto analysis ('80:20 rule') showed good results for easy identification of the top 20 % of critical modules.

Shatnawi et al. [5] has 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.

Jacek Czerwonka et al. [6] 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.

Mala, D. Jeya, and V. Mohan [7] has identified that Artificial Bee Colony model based test suite optimization generates near global optimal results and it converges within less number of test runs. In their approach they don't consider the critical components of the software. They had taken the path coverage for Unit testing. They also proved that the efficiency of ABC with the existing Genetic Algorithm based Test case optimization approach.

3. WORKING METHODOLOGY OF THE PROPOSED APPROACH

3.1 Criticality Analysis

Criticality analysis is the process of identifying the probability of failure modes of the component. The failure of a component can be made in two ways. One is to identifying how a component might get affected through other dependent components. Another one is to identify what type of failure that component might have. These are categorized as Sensitivity analysis and Severity analysis.

3.1.1 Sensitivity Analysis

Some components might be sensitive to other components. They may cause some serious effects on their dependent components. Sensitivity analysis calculates the probability value that might sensitive to other components. This process involves the following source code based coupling and cohesion measures.

- Fan-In – Number of other Components calling a given Component
- Fan-Out – Number of Components being called by a given Component.
- Information Flow – It represents the flow of data in collective procedures in the processes of a concrete system. It can be calculated as follows:

  Information Flow = (Fan-In * Fan-Out) ^{2} **(1)**

- Weightage of Methods in a class – This is used to show the complexity of a given Component by counting the number of independent paths in each of the methods of the given Component, whereas the Cyclomatic complexity [8] refers to the number of independent paths in a component.

  $$\text{Weightage of Methods} = \sum CCi \quad \textbf{(2)}$$

  Where,

  CC_i – Cyclomatic complexity of a method in a component, $\forall$ i=1 to m

  m – Total number of methods in a component.

- Weakness of Methods of a Component (WM) – The Weakness of methods of a component is the sum of Weakness of an each method. This can be represented as following:

  $$WM(Ci) = \frac{\sum Wmi}{m} \quad \textbf{(3)}$$

  Where,

  Wm_i - Weakness of a method in a component, $\forall$ i=1 to m

  m - Total number of methods in component

  The Weakness of a particular method (Wm_i) can be analysed using the following formula:

  $$Wm_i = [LV_i * V_i]^2 \quad \textbf{(4)}$$

  Here,

  $$LV_i = \frac{\text{Live variables count}}{\text{Number of Executable Lines}} \quad \textbf{(5)}$$

  $$V_i = \frac{\text{Live variables count}}{\text{Number of Variables}} \quad \textbf{(6)}$$

  Live variables means that the number of variable being used/live at a particular executable line. The following code snippet can be used to explain how live variables are counted.

  Table 1. Example Code Snippet

```
1   main()
2   void calculate()
3   {
4   int a, b, c;
5   read (a, b);
6   a=b + 5;
7   b=a - 5;
8   c=a + b;
9   write (a, b, c);
10  }
11  end;
```

  In the above, the lines 5, 6, 7, 8 and 9 are executable lines. So the executable lines count is 5. The calculation of live variables count is shown in the following table:

  Table 2. Counting Live Variables

Line No.	Live Variables	Count
5	a, b	2
6	a, b	2
7	a, b	2
8	a, b, c	3
9	a, b, c	3
Total		12

  From the above calculation, we have live variables count as 12 and total number of variables as 3. So, the weakness of method for the method void calculate () is calculated.

  $LV_i = 12/5 = 2.40$

  $V_i = 12/ \quad 3 = 4$

  $Wm_i = 9.60$

  $WM(C_i) = 9.60/1 = 9.60$

  The code contains only one method. So, the weakness of method for the sample code we taken is 9.60.

- Ratio of Pure Inherited Methods (RPIM) – It is the ratio of the number of pure inherited methods from a component by the dependent components.

  $$DRIM(C_i) = \frac{p}{m} \quad \textbf{(7)}$$

  Where,

  p - Number of pure inherited methods

  m - Total number of methods in an inherited component.

3.1.2 Severity Analysis

The Severity analysis is that the tactic of estimating the implications of failure and prioritizing the components per the severity level of

implications. The components with higher severity value could cause the functionalities of the system. To mitigate those failures, the high severity components will be tested fastidiously.

$$SV(Ci) = \sum_{k=1}^{p} SV\ (Mk\ (Ci)) \quad \textbf{(8)}$$

Where,

SV (M_k (C_i) – Severity Value of method k in the component C_i.

p – Number of methods in component C_i

SV (M_k (C_i)) is assigned with the values as **0.95, 0.75, 0.50, 0.25** based on the four characteristic described by Garousi [9].

Category 1: Criticality/ Importance of a message to the client. (The Control Flow of a method is purely depending on the outcome of a message from the server object called as Control Coupling.)

Category 2: Messages that call large amount of data/return values than other messages.

Table 3. Severity Analysis

S. No	Message Category	Severity Type	Severity Value
1.	Category 1	Catastrophic	**0.95**
2.	Category 2	Critical	**0.75**
3.	Category 3	Marginal	**0.50**
4.	Category 4	Minor	**0.25**

Category 3: The return values from methods might be used frequently or for critical decision/computation in the client than other messages.

Category 4: Some of the messages may be triggered more frequently than other messages.

3.1.3 Criticality Index

The critical value associated with each component is calculated based on Sensitivity and Severity metrics associated with it, the execution count and time taken by a component for its execution.

It can be calculated as follows:

$$CI\ (C_i) = P\ (CV\ (C_i)) * P\ (SV\ (C_i)) * P\ (E\ (C_i)) * \text{Time-taken}\ (C_i) \quad \textbf{(9)}$$

Where,

- P (CV (C_i)) - based on the Sensitivity Analysis of the Components.

$$P\ (CV\ (C_i)) = \frac{CV(Ci)}{\sum CV} \quad \textbf{(10)}$$

Here,

CV (Ci) – Sensitivity value of Component Ci.

CV – Sensitivity value of all components in SUT

- P (SV (C_i)) – based on the Severity Analysis of the Components.

$$P\ (SV\ (C_i)) = \frac{SV(Ci)}{\sum SV} \quad \textbf{(11)}$$

Here,

SV (Ci) – Severity value of Component Ci.

SV – Severity value of all the components in SUT

- P (E (C_i)) – based on execution count (E) of Component C_i. Fan-In count represents the execution count of the components.

$$P\ (E\ (C_i)) = \frac{FanIn(Ci)}{\sum FanIn} \quad \textbf{(12)}$$

Here,

Fan-In (Ci) – Fan-In count of Component Ci.

Fan-In – Fan-In count of all components in SUT

- Time-taken (C_i) - Total time taken by a component for its execution.

3.2 Artificial Bee Colony Algorithm

3.2.1 Introduction

Artificial Bee Colony algorithm is most recently defined by Dervis Karaboga, which was implied from the intelligent search behavior of honey bees. There are three groups of bees: employed bees, scouts bees and onlookers. We incorporate these groups of bees in our approach as the following agents [7].

- Search Agent – Employed Bee
- Selector Agent – Onlooker Bee
- Replace Agent – Scout Bee

These three agents work parallel to produce the optimized test cases by which the testing process can be reduced.

3.2.2 Algorithm of ABC

The ABC algorithm proposed in our approach is given below.

Step 1. Initialize the random test cases.

Step 2. Search agent applies the test cases in the executable state of the program and finds the fitness value of the test case.

Step 3. Selector agent evaluates the fitness value of test case. If the test case is not efficient, it produces the new test cases from the test case which has higher coverage value. It uses the following formula.

$$vij = xij + qij\ (xij - xkj) \quad \textbf{(13)}$$

Where,

k is a solution in the neighbourhood of I,

q is a random number in the range [-1, 1]

Step 4. The test cases with higher coverage value are stored in the repository.

Step 5. If no test cases are efficient, the Replace agent abandons the old test cases and produce new test case using the following formula.

$$xij = minj + rand(0,1)*(maxj - minj) \quad \textbf{(14)}$$

Step 6. Cycle=Cycle+1

Step 7. Repeat the steps 2-6 until the maximum cycle number is reached.

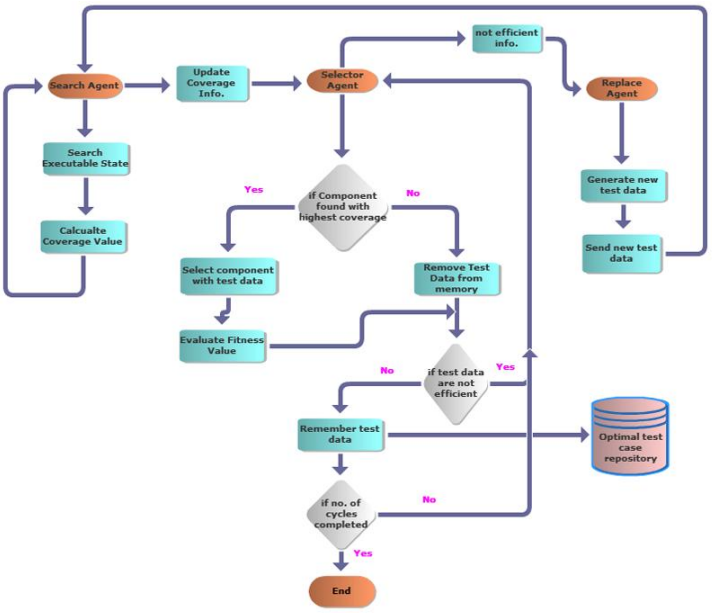

Figure 1. Framework of ABC Algorithm

3.2.3 Fitness Value Function of ABC

The Agents of ABC for generating new test cases are operated based on the fitness value of the test cases. The fitness value decides how much that test case is efficient. Here, we use Branch coverage [9] as fitness value function of ABC algorithm.

$$\text{Branch Coverage} = \frac{Numberofbranchescovered}{Totalbranches} * 100 \quad \textbf{(15)}$$

The test case with higher fitness value is stored in repository for further use.

3.3 Criticality Tester

The Criticality tester is the process which verifies the identified critical components of the system. The criticality tester is divided into two parts as Unit testing and Pair-wise testing.

3.3.1 Unit Testing

Unit testing is the testing technique considers a whole component as a unit and verifies its behavior is working fine or not. The test cases generated using ABC algorithm is applied to the unit testing. The unit testing considers Branch coverage as code testing technique. This process is continued until all the branches of the component are covered.

3.3.2 Pair-wise Testing

Pair-wise testing, also known as Integration testing, checks whether the functionalities of a component doesn't affect the integrated component. The pair-wise testing uses the connected components list extracted in initial phase, and the optimized test cases produced at the end of Unit testing.

4. IMPLEMENTATION OF THE PROPOSED APPROACH

To automate the proposed approach, we implements a tool "JCTester – Java Components Tester" using Java Swings environment. To illustrate the proposed approach, we had taken "Blood Bank Management System" as a case study. The screen shots are included in the Appendix. The Blood Bank Management System (BBMS) has 12 components.

4.1 Sensitivity Analysis

The Sensitivity Analysis extracts the coupling and cohesion metrics as Fan-In, Fan-Out, Information Flow, Weightage of Methods in a Component, Weakness of Methods of a Component and Ratio of Pure Inherited Methods from the components. The metrics value for all the components of BBMS are listed in the following the table.

Table 4. Sensitivity Analysis of Components

Class Name	Fan-In	Fan-Out	INFO	Weightage of Methods	Weakness of Methods	RPIM
Admin	1	4	16	1.17	4	0
Attendance 1	1	1	1	1.28	3	0
Camp	1	2	4	1.84	5	0
Collect	1	1	1	0.81	5	0
Donar	3	0	0	0.8	9	0
Employee	2	2	16	2.11	5	0
Equip	1	1	1	0.6	5	0
Issue	1	1	1	0.86	5	0
Recepeint	1	2	4	1.35	5	0
Request	1	1	1	1.1	5	0
Salary1	1	1	1	1.36	3	0
Stock	3	1	9	0.47	5	0

4.2 Severity Analysis

Based on the failure type of a component, the severity analysis categorizes each component of BBMS into Catastrophic, Critical, Marginal and Minor.

For example, consider the following code having catastrophic severity type.

```
public class Stock {
            ---code----
Issue req = new Issue ();
public void Stockdetails (int ridd, String redd, String bgg, int qtt) {

if ( req.issueno() > 2000 ) {

        System.out.println ("we can't process this order");
}
          --- code ---
```

Figure 2. Component having catastrophic type of Severity because the method call is resides in the conditional statement

At the end of this, we have the following table describes the severity type of all components in BBMS.

Table 5. Severity Analysis of Components

Class Name	Severity Type	Value
Admin	Critical	0.75
Attendance1	Minor	0.25
Camp	Critical	0.75
Collect	Minor	0.25
Donar	Minor	0.25
Employee	Critical	0.75
Equip	Minor	0.25
Issue	Minor	0.25
Recepeint	Critical	0.75
Request	Minor	0.25
Salary1	Minor	0.25
Stock	Catastrophic	0.95

4.3 Criticality Index

In addition to the results of sensitivity analysis and severity analysis, the components time taken to execute and the execution count of each component is considered while calculating the criticality index of the components. Based on criticality value, the components are prioritized and listed in the following table. The component with higher critical value is the most critical component to the system.

Table 6. Critical value of Components

Class Name	Critical Value
Admin	0.922
Employee	0.39
Stock	0.174
Camp	0.106
Recepeint	0.046
Donar	0.021
Collect	0.02
Request	0.01
Issue	0.01
Equip	0.006
Attendance1	0.005
Salary1	0.003

Having the critical value as higher, the Component Admin is most critical to the case study we taken, "Blood Bank Management System".

4.4 Unit Testing

4.4.1 Code Instrumentation

The Unit testing is performed based on the Branch Coverage of the source code. To track which branches are covered, we instrumented some additional code in the existing source code that doesn't affect the original functionalities of the component. The process of code instrumentation is shown in the following figure 3 for the method **Equipdetails** in the class **Equip**.

```
public void Equipdetails(int idd, String eqq, int itnn, String pdd)
throws Exception {
File file1 = new
File("src//instrumented//bloodbank1.Equip.Equipdetails.txt");
FileOutputStream fos = new FileOutputStream(file1, false);

if (eqq.length() <= 0 || pdd.length() <= 0) {

fos.write("B1\n".getBytes());
System.out.println("Invalid Details");
}
---other code---
}
```

Figure 3. Instrumented Code in the method Equipdetails

4.4.2 Test Case Generation using ABC

For the method shown in figure 3, we illustrate how test cases are generated using the ABC algorithm. The initial random test cases of the above method are given in the following table.

Table 7. Initial random population of Test Case (T1)

Parameter Type	Parameter Value
int	1584724267
class java.lang.String	rd
Int	-466289813
class java.lang.String	null

The Search agent of ABC, evaluates the fitness value of the initial test case. The method Equipdetails contains only one branch, but the test case is not able to cover the branch. So, the Search agent evaluates the fitness of T1 as 0%. This fitness value is passed to the Selector agent and it generate new test case as follows.

vij = xij + qij (xij - xkj)

Here X is the initial test case generated (T1). V is the new test case produced by the Selector agent.

$V_1=X_1+q(X_1-X_4)$ = 1584724267+-1(1584724267-null)

V_1=1290994490

Likewise,

V_2=fd

V_3=2036439598

V_4=

Table 8. New Test Case (T2) generate by Selector Agent

Parameter Type	Parameter Value
int	1290994490
class java.lang.String	fd
Int	2036439598
class java.lang.String	

The test case T2 generated by Selector Agent is able to cover the branches of the method Equipdetails. Hence the fitness value is 100%, the generated test case is stored in the repository.

If the Selector agent doesn't able to produce efficient test cases, the Replace agent take care of producing new set of test cases. This process will continue until all the branches are covered or the maximum test cycle is reached. In order to generate test case from different data types, we use ASCII value for char data type and radix value for String data type.

4.5 Pair-wise Testing

Pair-wise ensures that the functionalities of a component don't get affected by its integrated components. The optimized test cases that cover entire component are passed from Unit testing to perform the pair-wise testing. Our approach ensures whether my test case is able to cover the integrated components.

Table 9. Example of Pair-wise Testing

Source Class	Method that invoke the Integrated Class	Destination Class	Method that invoke the Integrated Class
Employee	Employeedetails	Recepient	RecepientDetails
Employee	Employeedetails	Admin	Validate
Employee	Employeedetails	Stock	Sno

By invoking the component Employee, the test case covers its integrated components Recepient, Admin, Stock. This ensures that the functionalities of the integrated component are working fine and it doesn't affect the other components.

5. EVALUATION AND COMPARISON WITH GA

We proposed this approach to identify and verify the critical components of the SUT. The main objective is to reduce the testing time of the Critical components. So, we incorporate the optimized algorithm, Artificial Bee Colony. To prove that ABC is better, we compared it with another known optimization algorithm Genetic Algorithm.

5.1 Genetic Algorithm

Genetic Algorithm is the evolution of natural species in searching optimal solution. The Genetic algorithm is also applied for test case optimization. The test cases are represented as chromosomes which include methods to be invoked and the values passed. The Genetic algorithm includes the functions like Cross over, Mutation to produce new optimized test cases [9]. The Genetic algorithm also uses Branch coverage as its fitness value function.

5.1.1 Test Case Generation using Crossover

Crossover selects a random index in the test population say m, and divides the population into two different portions as 0…m-1 and m…n. The divided portions are recombined as m…n and 0…m-1 and produces new set of test case population

Say for example, for the variables a, b and c

Test Case 1: 1, 2, 1
Test Case 2: 1, 1, 1

After cross-over at the **second** position, we get new test cases as follows:

Test Case 11: 1, 1, 1
Test Case 21: 1, 2, 1

This new generation of test cases is then evaluated based on their effectiveness and then either selection or removal will be done..

5.1.2 Mutation

Mutation changes the member of test case population and reproduces the remaining members for new test case population.

Test Case 1: 1, 2, 1
Test Case 2: 1, 1, 1

After mutation operator is applied to first member of these test cases, we will get the new generation of test cases like,

Test Case T1: 2, 2, 1
Test Case T2: 1, 1, 2

Now this new generation of test cases will be evaluated and the procedure is repeated.

5.1.3 Pseudocode of Genetic Algorithm

The pseudocode of the Genetic Algorithm is given as follows.

Step 1. Generate Initial set of Test case population

Step 2. Evaluate the fitness value of the population.

Step 3. Select best test case that has higher fitness value.

Step 4. Perform Crossover and Mutation to produce new test cases.

Step 5. Evaluate the fitness value of new population.

Step 6. Replace the worst test case population with lower fitness value.

Step 7. Repeat steps 3 to 6 until maximum test cycle number is reached.

5.2 Case Studies

The proposed approach has been compared against GA with the help of 5 industrial case studies. The information of the case studies are given in the following table.

Table 10. Case Study Information

S. No	Case Study Name	Case Study ID
1	Blood Bank Management System	CS1
2	Banking Management System	CS2
3	Apache Ant Tool 1.2	CS3
4	Library Management System	CS4
5	Hospital Management System	CS5

5.3 Comparison

The specified case studies are applied on both the approaches and the results are compared as follows. The following table shows the comparison of time taken to test the components.

Table 11. Comparison of Time Taken

S. No	Case Study ID	ABC Algorithm (in sec)	Genetic Algorithm (in sec)
1	CS1	4.63	11.875
2	CS2	13.35	26.953
3	CS3	17.1	35.142
4	CS4	40.23	75.99
5	CS5	211.74	589.22

The time taken to complete the testing process is compared using the following graph.

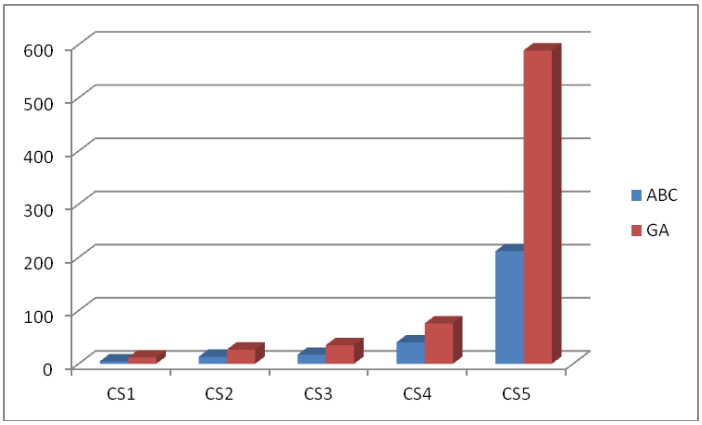

Figure 4. Graph to compare Testing Time taken

The branch coverage information and the test cases needed are given in the following table. Most of the time, the coverage doesn't reaches100% due to infeasible branches in the code.

Table 12. Comparison of Coverage Information

S. No	Case Study ID	ABC Algorithm		Genetic Algorithm	
		Coverage %	Test Cases	Coverage %	Test Cases
1	CS1	95	242	56	413
2	CS2	98	365	43	758
3	CS3	85	322	50	635
4	CS4	99	611	69	1543
5	CS5	98	745	80%	1759

The Branch coverage value of each case study between ABC and GA is compared using the following graph.

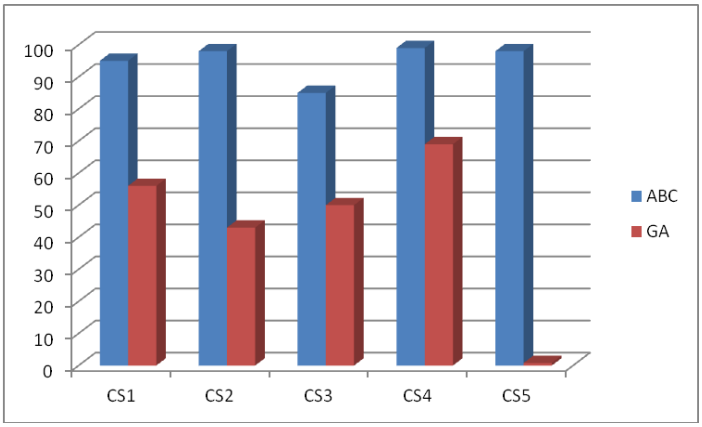

Figure 5. Graph to compare Branch Coverage value

The test cases neeeded to cover all the braches of code between ABC algorithm and Genetic Algorithm are compared using the following graph.

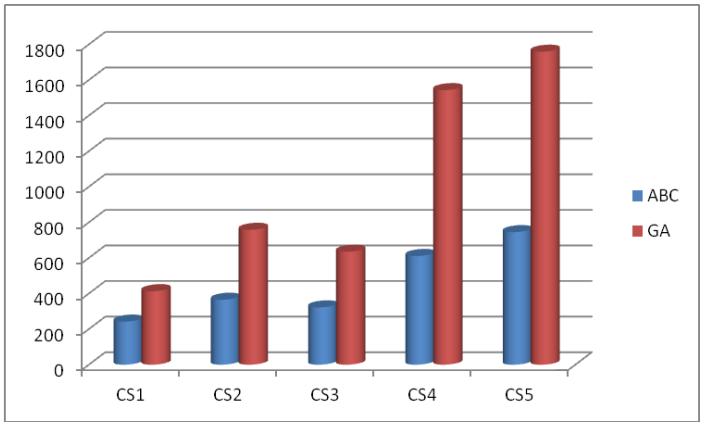

Figure 6. Graph to compare Test Cases needed

From the above comparisons, we came to know that the coverage value of GA is not consistent even the number of test cases generated are increased. The time taken to complete the testing process is also large than the Artificial Bee Colony algorithm. So, it is proven that the test case optimization using ABC is better than the Genetic Algorithm.

6. ACKNOWLEDGMENTS

The proposed work is a part of UGC funded major research project "Critical components Identification and Verification for Real time Complex System using Artificial Bee Colony based optimization Approach".

7. CONCLUSION

The proposed work has been automated to identify and verify the critical components of any real time complex system. As stated earlier, the objective of reducing the process of testing the critical components is achieved with the help of Artificial Bee Colony algorithm. It is proved that the optimization using ABC is better than the existing Genetic algorithm approach.

8. REFERENCES

[1] Ohlsson, Niclas, M. Helander, and C. Wohlin. 1996. Quality improvement by identification of fault-prone modules using software design metrics. In *Proceedings of the International Conference on Software Quality*. 1-13.

[2] Hendry, Sallie, Kafura Dennis. 1981. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering.* Vol. 7, No. 5. 510-518.

[3] Garousi, Vahid, Lionel C. Briand, and Yvan Labiche. 2006. Analysis and visualization of behavioral dependencies among distributed objects based on UML models. *Model Driven Engineering Languages and Systems.* (Springer Berlin Heidelberg). 365-379. 2006.

[4] Christof Ebert. Classification techniques for metric-based Software development. *Software Quality Journal.* Vol.5, No.4. 255-272.

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

[6] Czerwonka, Jacek, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. 2011. Crane: Failure prediction, change analysis and test prioritization in practice--experiences from windows. In *IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), 2011*. 357-366.

[7] Mala, D. Jeya, and V. Mohan. 2009. ABC Tester—Artificial Bee Colony Based Software Test Suite Optimization Approach. *International Journal of Software Engineering* 2, no. 2. 15-43.

[8] Aditya P Mathur. Foundations of Software Testing. Pearson Education India. ISBN: 9788131707951.

[9] Pargas, Roy P., Mary Jean Harrold, and Robert R. Peck. 1999. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability* 9.4 (1999): 263-282.

APPENDIX

The screen shots and the graphs generated by JCTester are included here.

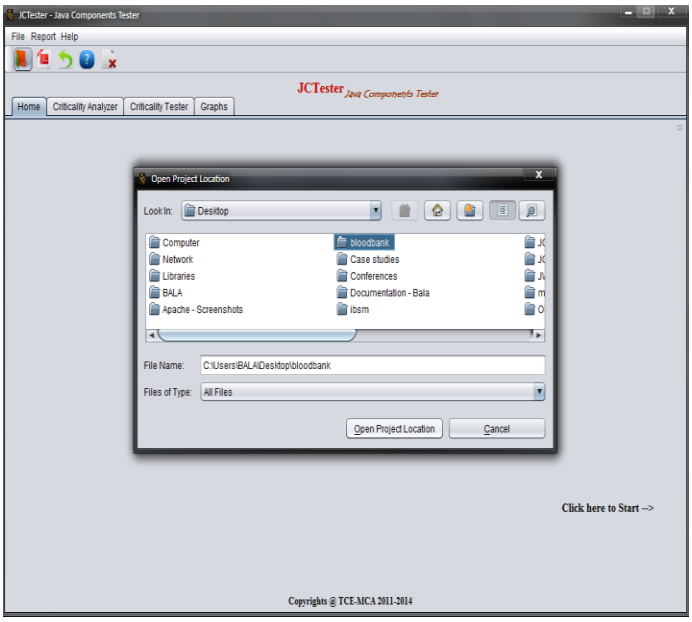

Figure 7. Selecting the project location to be tested

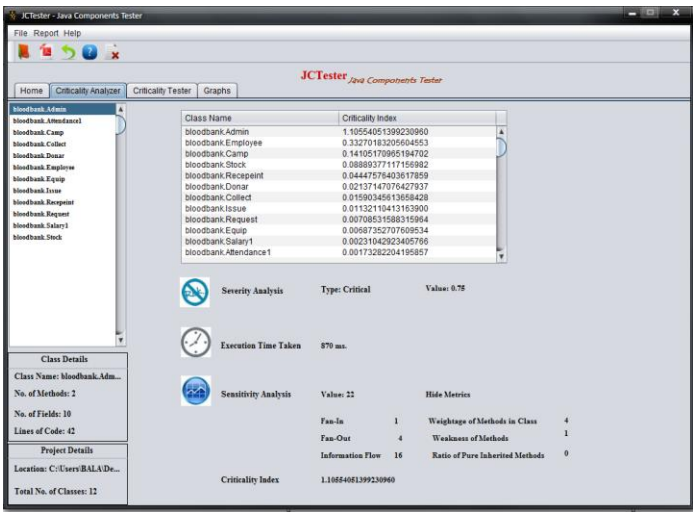

Figure 8. Extracting Metrics and Prioritize the Critical components based on Critical value

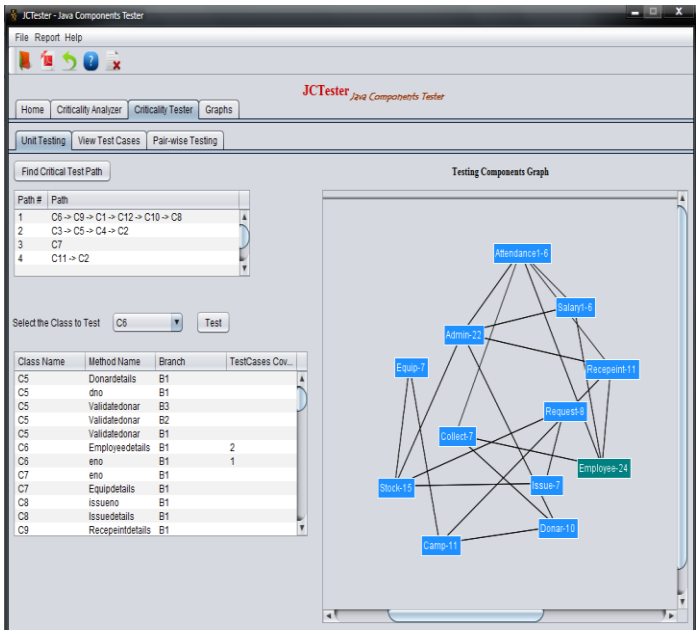

Figure 9. Unit Testing of Components and the Graph get updated after the component Tested

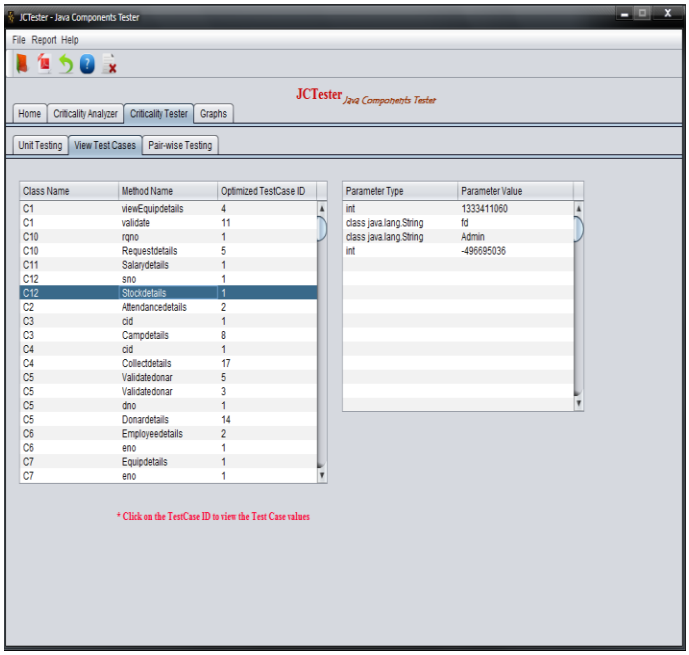

Figure 10. View Optimize Test Case values

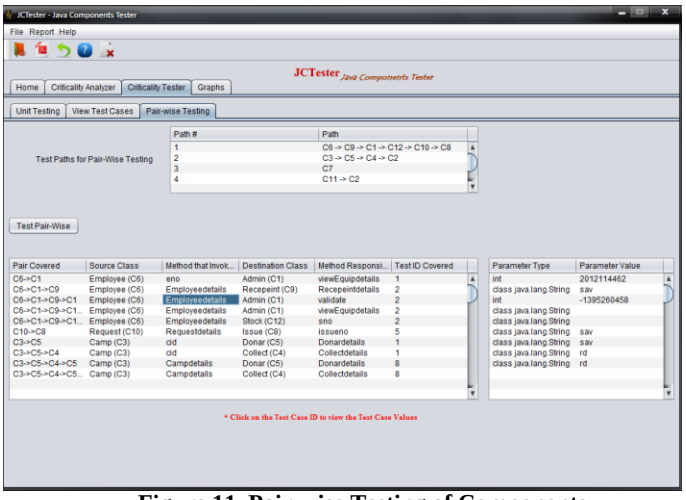

Figure 11. Pair-wise Testing of Components

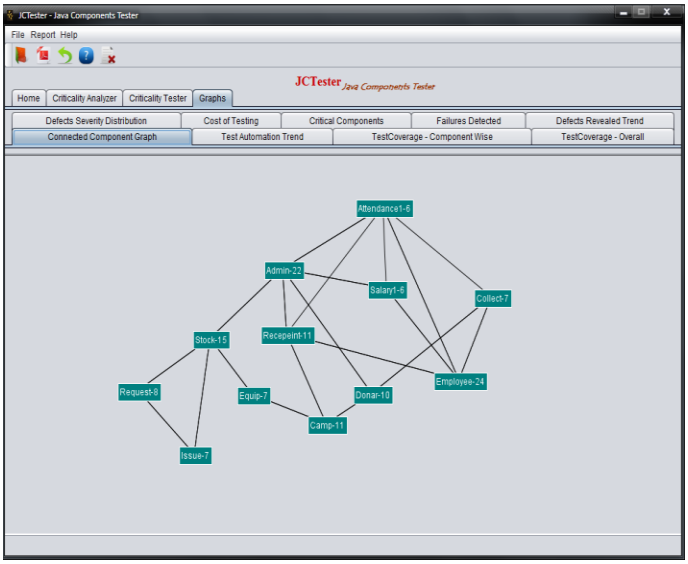

Figure 12. Connected Component Graph shows the dependency of the components

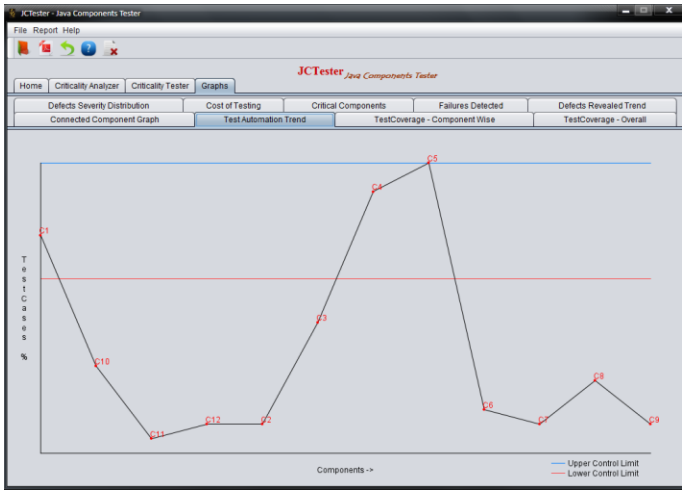

Figure 13. Test Automation Trend Graph shows which components are crossed the Upper limit of Test cases count

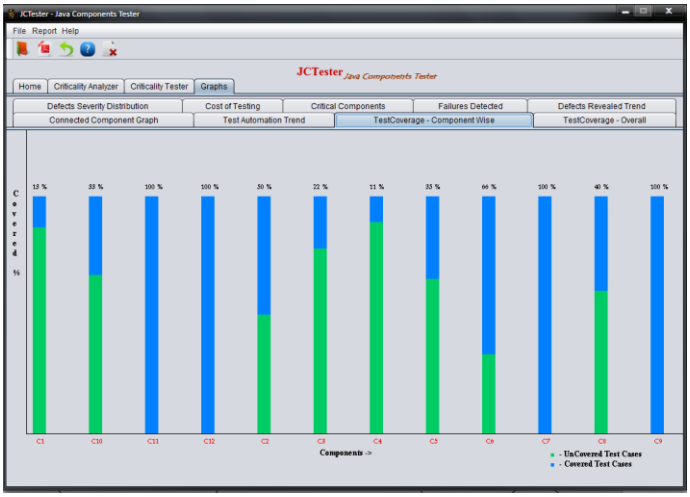

Figure 14. Test Coverage Component wise Graph shows how much percentage of test cases covered for each component

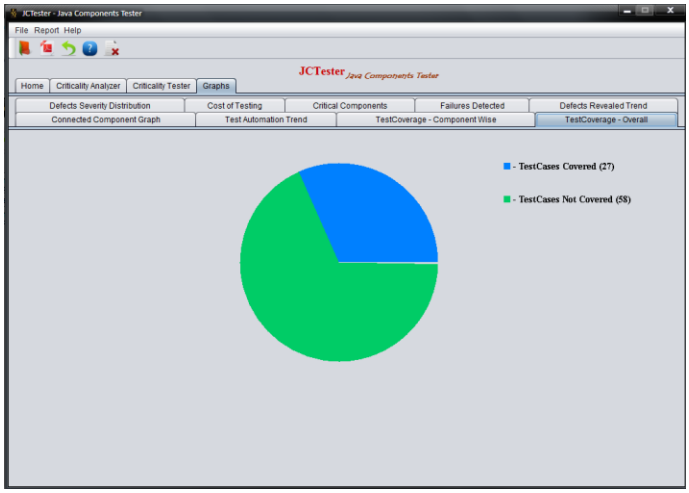

Figure 15. Test Coverage Overall Graph shows how much test cases are covered in all the components.

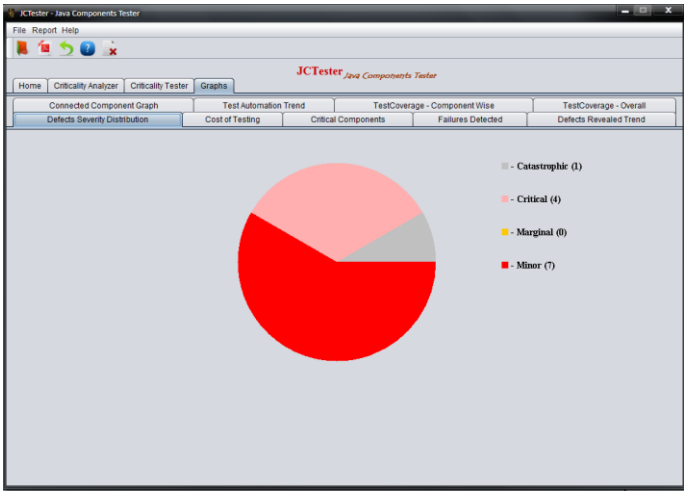

Figure 16. Defects Severity Distribution shows the count of components against each category of Severity type

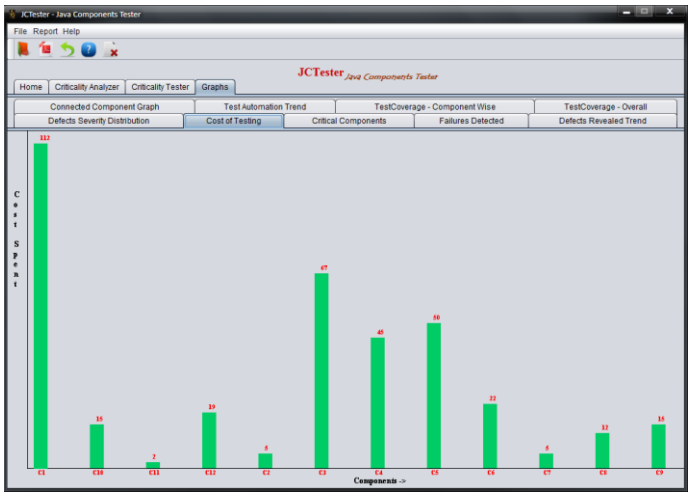

Figure 17. Cost of Testing Graph shows how much cost needed to test each component

Figure 19. Failures Detected Graph shows how much test cases are failed to cover the components on each test cycle

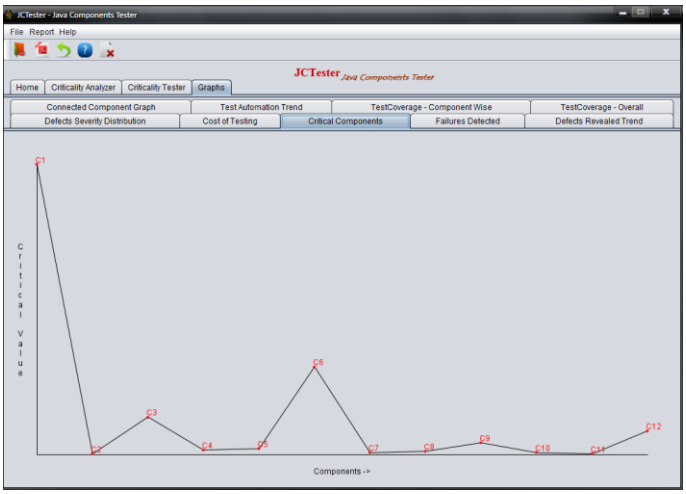

Figure 18. Critical Component Graph shows which components are most critical to the system

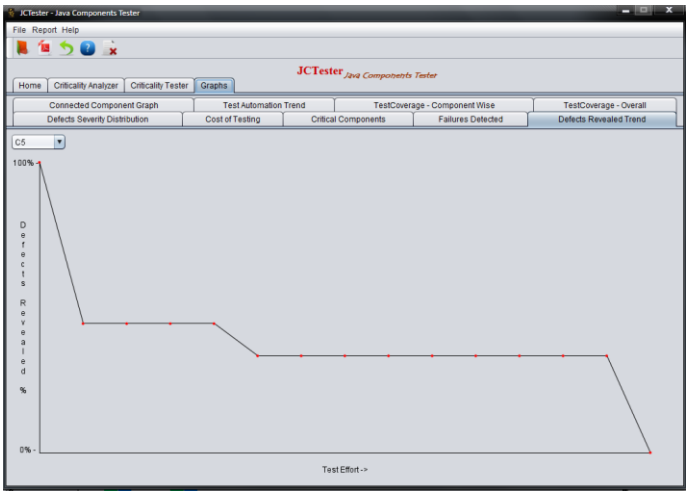

Figure 20. Defects Revealed Trend Graph shows how branches are covered throughout the test cycle