Int. J. Advance. Soft Comput. Appl., Vol. 1, No. 1, July 2009
ISSN 2074-8523; Copyright © ICSRS Publication, 2009
www.i-csrs.org

Automatic Test Case Generation for UML Object diagrams using Genetic Algorithm

M. Prasanna[1] and K.R. Chandran[2]

[1]Research Scholar, Department of CSE,
PSG College of Technology, Coimbatore 641 004, India
e-mail: mp_psg@rediffmail.com

[2]Professor, Department of IT,
PSG College of Technology, Coimbatore 641 004, India
e-mail: chandran_k_r@yahoo.co.in

Abstract

A new model based approach for automated generation of test cases in object oriented systems has been presented. The test cases are derived by analyzing the dynamic behavior of the objects due to internal and external stimuli. The scope of the paper has been limited to the object diagrams taken from the Unified Modeling Language model of the system. Genetic Algorithm's tree crossover has been proposed to bring out all possible test cases of a given object diagram. Illustrative case study has been presented to establish the effectiveness of our methodology coupled with mutation analysis

Keywords: *Depth First Search, Object Diagram, Software Testing, Test Case, UML.*

1. Introduction

Software testing [1] is an important activity in software development life cycle. Software organizations spend considerable portion of their budget in testing related activities. A well tested software system will be validated by the customer before acceptance. Testing includes executing a program on a set of test cases and comparing the actual results with the expected results. Testing should also focus on fault prevention. Test cases are usually derived from software artifacts such as specifications, design or the implementation. To test a system, the implementation must be understood first which can be done by creating a suitable model of the system.

A common source for tests is the program code. Every time the program is executed, the program is tested by the user. So we have to execute the program with the specific intent of fixing and removing the errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques. UML [Unified Modeling Language] [19] is a widely accepted set of notations for modeling object oriented system. It has various diagrams for depicting the dynamic behavior of objects in a system. In this paper we use Object diagrams represented in the form of a tree to extract test cases to verify/validate the behavior of objects concerned.

2. Related Works

Emanuela et al [3] have proposed a model based testing techniques with test cases generated from UML Sequence diagrams. Sequence diagram is translated to Labeled Transition Systems and test cases are generated for mobile phone Applications. Test case works perfectly for small sized mobile phone application. For bigger applications with bigger LTSs, the set of test cases is greater and application functionalities test coverage is still a problem. Minimizing test case redundancy also needs to be dealt using their approach. Samuel et al [5] have proposed automatic test case generation for UML state diagrams. It covers all the events associated with state diagrams. They have reduced the number of test cases by testing the borders determined by simple predicates. They have illustrated their test case automation for an ice cream vending machine. They were not able to achieve globally optimal solution using alternating variable method. They suggested genetic algorithm to achieve the same. Monalisa et al [4] presented use case diagram graph and Sequence diagram graph for generating test cases from use case and sequence diagrams for a PIN Authentication scenario in an ATM system. Test case covers use case initialization faults, dependency faults and operational faults. It checks the sequential dependency that may exists among use cases. If test data for test case is not integrated, then it will lead to mine the same data repeatedly. They concentrated only on system level testing. Iftikhar[6] has proposed an object oriented approach to convert UML class and state chart diagram for a Dishwasher system into Java code and his thesis result shows that it is 60 % more efficient and 3 times more compact than that of Rhapsody's approach. Source code is automatically generated from the given UML class and state chart diagrams. Shaukat Alia et al [8] presented a technique that combines UML collaboration diagrams and statecharts to automatically generate an intermediate test model, called SCOTEM (State COllaboration TEst Model) to generate valid test paths. Their results show that the proposed technique effectively detects all the seeded integration faults. Mainly All-Path Coverage is very expensive and it can scale up in all situations. Effectiveness of their algorithm is proven with Stack case study. Many researchers and practitioners have been working in generating optimal test cases based on the specifications.

3. The Proposed Method

Firstly, object diagram is drawn using rational rose software. It shows a snapshot of the detailed state of the system at a point in time. Secondly, it is mapped to a tree with root node and child nodes. Thirdly, we apply Genetic Algorithm's cross over operator which yields in new generation of trees. New generation of trees are converted into binary trees. Depth first search technique is applied on the binary trees which results in test case set. Since we use depth first search algorithm for generating test cases, no single path can be revisited there by eliminating redundancy.

Our proposed methodology involves the following steps:

1. Construct object diagram using rational rose software and store it with .mdl as extension.
2. Parse the .mdl file and capture the object names.
3. Build a tree using object names and apply genetic algorithm's cross over technique.
4. New generation of trees are formed and convert it to binary trees.
5. Traverse new generation of binary trees using Depth First Search technique.
6. All the valid, invalid and termination sequences of the application can be obtained using Step 5.

The above steps are illustrated in the form of flowchart as shown in Figure 1.

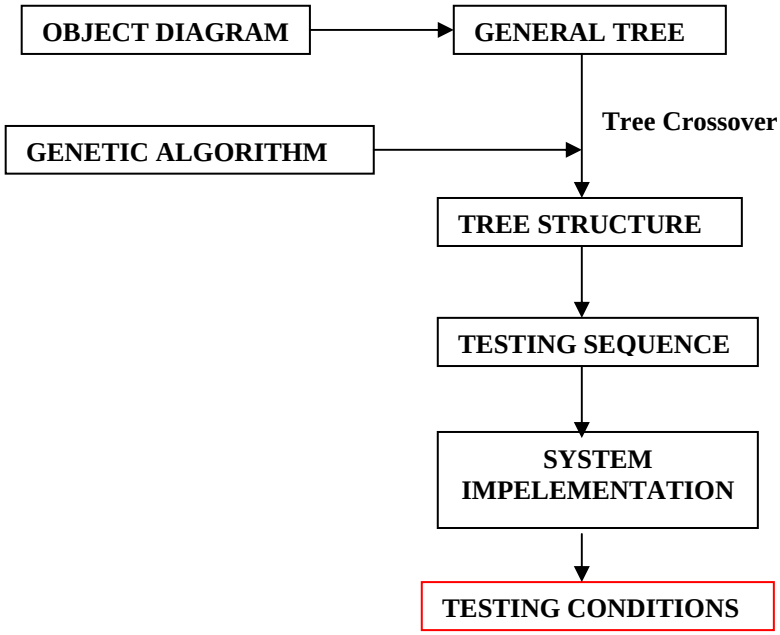

Figure 1: Flowchart of Proposed Methodology

4. Case Study

An object diagram of a banking system created using Rational Rose tool has been considered for test case automation process. In this scenario, the user initiates the process by entering bank name, user name and password in the Bank System (BS) object and authentication takes place. For new users, the system invokes New_User_form ( ) and collects details like name, location, phone number, account type, amount, and customer ID and it is updated to the database. Already existing users can perform banking operations like deposit, withdraw, balance and so on. We now generate the possible test cases for this problem using our proposed methodology.

Step 1:

The Object diagram for a banking system is shown in Figure 2. It represents the dynamic behavior of objects in a banking system.

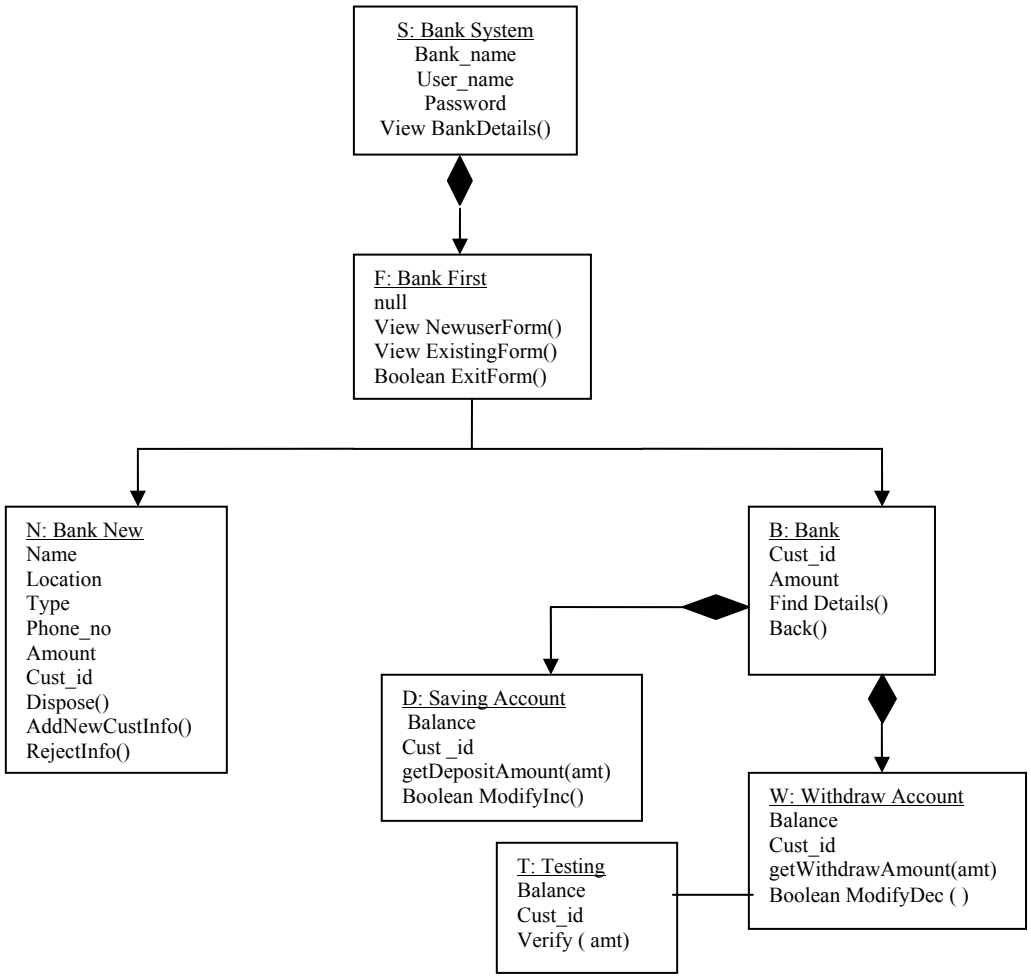

Figure 2: Object Diagram of Banking System

Step 2:

Tree Form:

Tree structure is very simple, understandable and can be easily maintained in the computer memory. We could easily traverse the tree to obtain complete Test case set. Redundant test cases can be avoided and time complexity is less in trees. To represent banking system in a tree form, we need to make the following modifications as illustrated in Figure 3a:

a. The objects are represented as nodes and placed in a vertical line one after the other
b. The object inputs (attributes) are arranged in left branch of the corresponding node
c. The object outputs (methods) are arranged in the right branch of the corresponding node
d. If any duplication occurs, then object name is added as prefix for the nodes

The steps (a to d) discussed above also applies to the figures 3b to 3d.

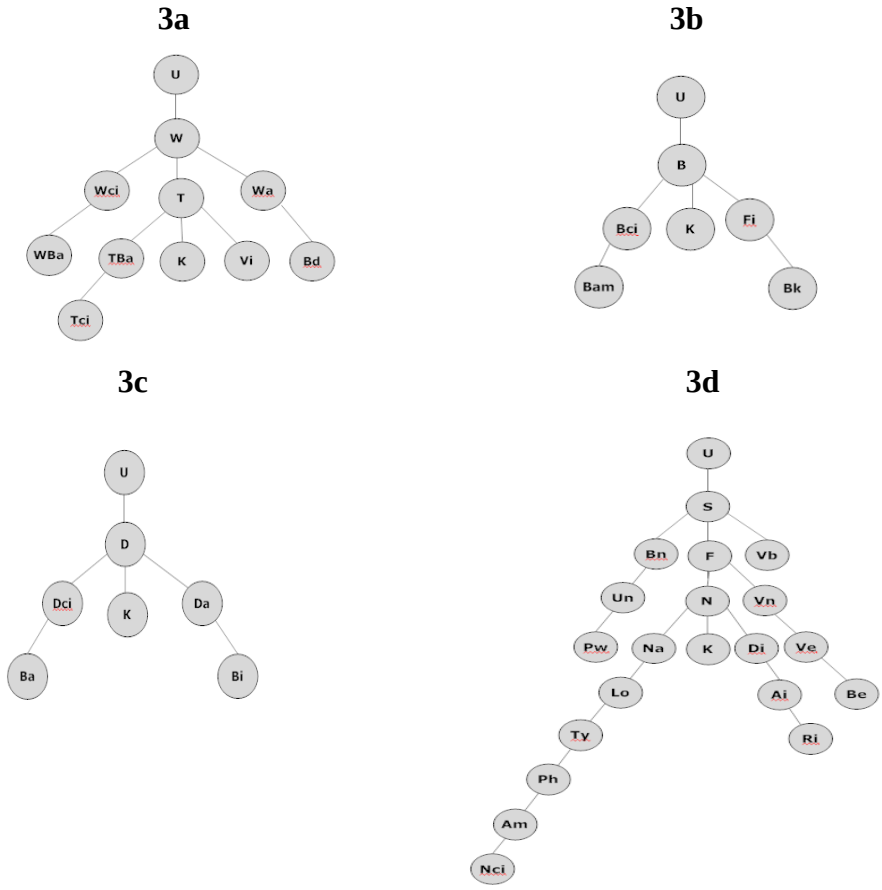

Figure 3: Converting Object to Tree Structure

Step 3:

Genetic Algorithm's cross over method is applied on the trees shown in Figure 3a to 3d. One crossover point is selected among the parent trees and it gives new generation offspring (Figure 4). This technique is applied on all the tree structures obtained from the object diagram of the application. Figure 5 is also obtained in the same manner.

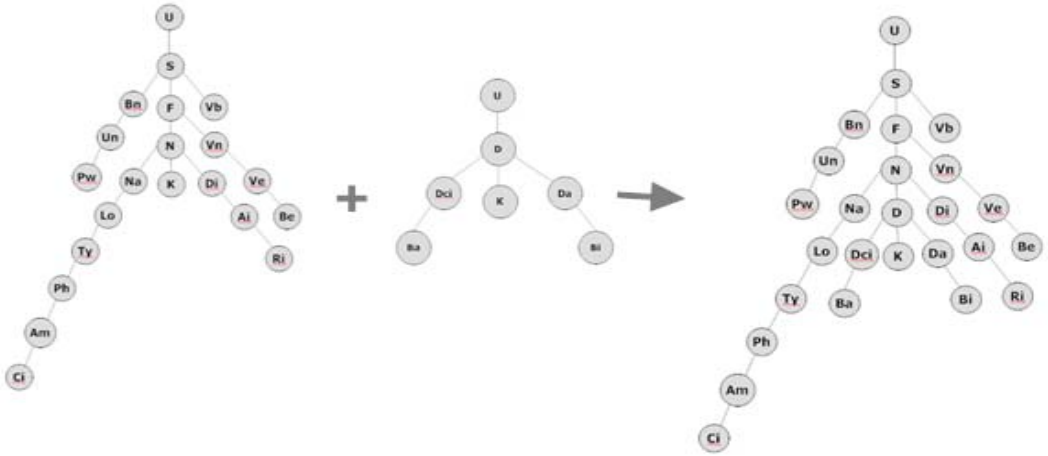

Figure 4: Tree Cross Over (3d & 3c) in Banking System

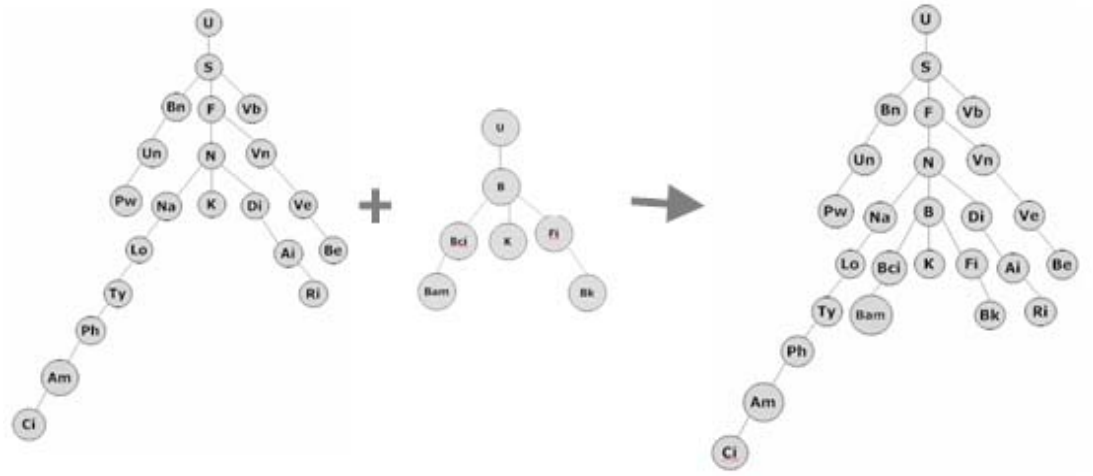

Figure 5: Tree Cross Over (3d & 3b) in banking System

Step 4:

The tree shown in Figure 4 is not a binary tree. The binary tree can be formed by arranging the nodes in the left branch of the root node in vertical order and arrange its sibling in horizontal order. Now this will form a binary tree and it

is shown in figure 6a. The structure is redrawn to depict the exact binary tree as shown in figure 6b. Similarly we use figure 7 to represent the generation of the binary tree from figure 5 using step4.

6a **6b**

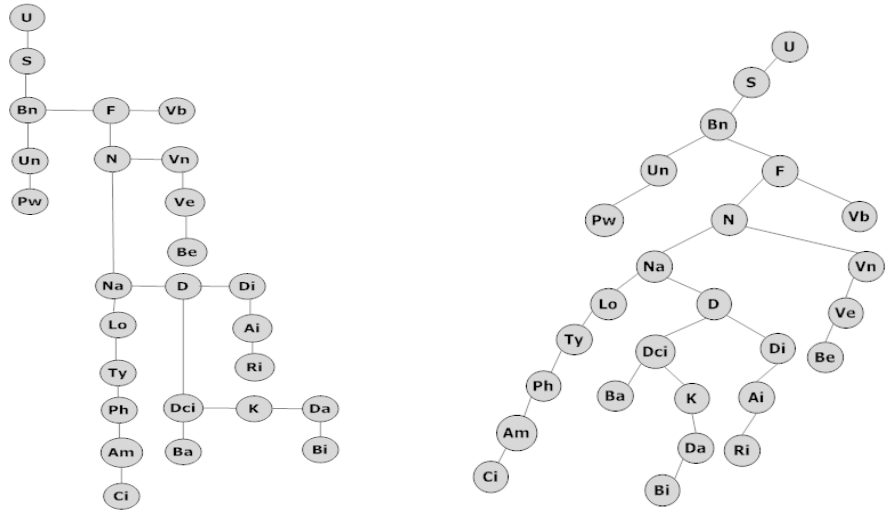

Figure 6: Binary Tree form of banking System by crossing fig 3d & 3c

7a **7b**

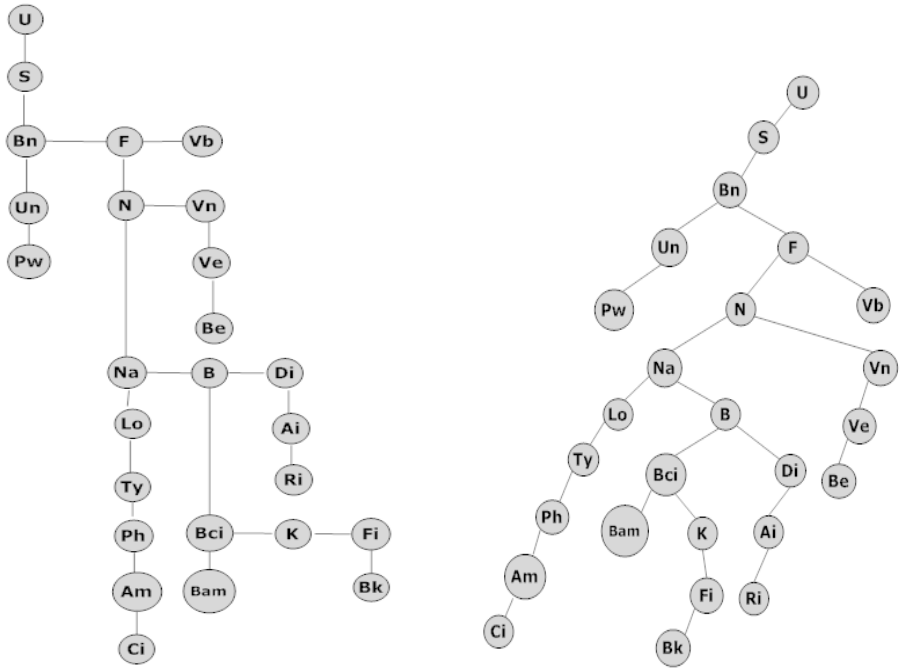

Figure 7: Binary Tree form of banking System by crossing 3d & 3b

Step 5:

Traverse the binary tree using Depth First Search technique. It gives all the valid, invalid and termination sequences for the given application. The mapping information is given in Table 1 and the testing sequences of the banking system are shown in the Table 2.

Table 1: Mapping Information

S. No	Objects in sequence model	Nodes
1.	User	U
2.	Deposit	D
3.	Bank First	F
4.	Banking System	S
5.	Bank	B
6.	Withdraw	W
7.	Testing	T
8.	Bank Name	Bn
9.	User Name	Un
10.	Password	Pw
11.	Name	Na
12.	Location	Lo
13.	Phone	Ph
14.	Amount	Am
15.	Customer ID	Ci
16.	View Bank	Vb
17.	View New User Information	Vn
18.	Bank Customer ID	Bci
19.	Bank Amount	Bam
20.	Deposit Customer ID	Dci
21.	Boolean Modify Dec	Bd
22.	Withdraw Customer ID	Wci
23.	Withdraw Balance	WBa
24.	Testing Balance	TBa
25.	Testing Customer ID	Tci
26	Boolean Modify Inc	Bi
27.	Balance	Ba
28.	View Existing	Ve
29.	Boolean Exit	Be
30.	Dispose	Di
31.	Add Information	Ai
32.	Reject Information	Ri
33.	Find Information	Fi
34.	Back	Bk
35.	Deposit Amount	Da
36.	Withdraw Amount	Wa
37.	Verify	Vi
38.	Type	Ty
39.	End	K

Table 2: Test Case Table for Banking System

S.No	SEQUENCE	RESULT
1.	USBnUnPw	VALID
2.	USBnFVb	VALID
3.	USBnFVnVeBe	VALID
4.	USBnFNNaDDiAiRi	VALID
5.	USBnFNNaWDiAiRi	VALID
6.	USBnFNNaBDiAiRi	VALID
7.	USBnFNNaWWciWba	VALID
8.	USBnFNNaBBciBam	VALID
9.	USBnFNNaDDciKDaBi	VALID
10.	USBnFNNaBDciKFiBk	VALID
11.	USBnFNNaLoTyPhAmCi	VALID
12.	USBnFNNaWTWaBd	VALID
13.	USBnFNNaWTTbaTci	VALID
14.	USDDiAiRi	INVALID
15.	USBnFNNaWTTbaKVi	VALID
16.	USBnVbVeBe	INVALID
17.	USTWaBd	INVALID
18.	UNNaDCiBa	INVALID
19.	UBBciWBa	INVALID
20.	USNaLoTyPhAmCi	INVALID
21.	UTWaBd	INVALID
22.	UWBk	INVALID
23.	UDiAiRi	INVALID
24.	USBnVnPwFNNaLoTyPhAmCiDDciBaKDaBiDiAiRiVnVeReVb	TERMINATION
25.	USBnVnPwFNNaLoTyPhAmCiBBciBamKFiBkDiAiRiVnVeReVb	TERMINATION
26.	UBBciBamDDciBaKBaBiFi	TERMINATION
27.	UBBciBamWWciWBaTTBaTciKViWaBdFiBK	TERMINATION
28.	UDDciBaWWciWBaTTBaTciKViWaBdDaBi	TERMINATION
29.	USBnUnPwFNNaLoTyPhAmCiWWaWbaTTbaTciK ViWaBdDiAiRiVnVeBeVb	TERMINATION
30.	UBBciBam	VALID
31.	UDDciBa	VALID
32.	UBBciWWciWBa	VALID
33.	UDDciWWciWBa	VALID
34.	UBBciWFiBK	VALID
35.	UDDciWBaBi	VALID
36.	UBBciWWciTTBaTci	VALID
37.	UDDciWWciTTBaTci	VALID
38.	UBBciWWciTTBaKVi	VALID
39.	UBBciWWciTWaBd	VALID
40.	UDDciWWciTWaBd	VALID
41.	UBBciBam	VALID
42.	UBDDciBa	VALID
43.	UBDDciKBaBi	VALID
44.	UBDFi	VALID
45.	UDDciBa	INVALID
46.	UKBaBi	INVALID
47.	UDDaBi	INVALID

5. Mutation Testing

The effectiveness of test cases can be evaluated using a fault injection technique called *MUTATION ANALYSIS*. Mutation testing is a process by which faults are injected into the system to verify the efficiency of the test cases. Mutation based analysis is a fault-based testing strategy that starts with a program to be tested and makes numerous small syntactic changes into the original program. Program with injected faults is called MUTANTS. The faults are inserted and tested in the following manner. One faulty version of the program is created at a time and run against all the test cases one by one until either fault is revealed or all test cases are executed. A fault is considered to be revealed, if the output of faulty version of program is different from the original program on same input. If a test case set is capable of causing behavioral differences between original program and mutant, mutant is considered as killed by test. The product of mutation analysis is a measure called Mutation Score, which indicates the percentage of mutants killed by a test set. Mutants are obtained by applying mutation operators that introduce the simple changes to original program (or Specification).The faults are kept in separate versions of the program to avoid interactions between faults such as masking.

5.1 Fault Injection

The test cases derived using the Genetic Algorithm for the Banking system table 1 is considered for testing process. The following parameters listed in table 3 were considered for mutation analysis process. For the Banking system object diagram, we created 61 mutants that use mutation operator as shown in Table 3. The summary of the mutants are shown in Table 4.

Table 3: Operator and Description

S.No.	OPERATOR	DESCRIPTION
1	Function	Replaces the name of the function
2	Guard condition	Changes/deletes the guard condition
3	Relation operator	Replaces the relational operator
4	Data value	Replaces the value of data
5	Data name	Replaces the name of data
6	Parameter	Change the letters of the parameter
7	SQL query	Change the query lines and field
8	Subclass name	Change the super class name in the sub class

Table 4: Summary of the mutants for Banking System

Operator	Faults Injected	Faults Found
Function	11	11
Guard condition	3	2
Relational operator	8	5
Data value	15	10
Data name	5	5
Parameter	3	3
SQL query	8	6
Subclass name	7	7
Total	**61**	**49**

5.2 Mutation Score

The product of mutation analysis is a measure called Mutation Score, which indicates the percentage of mutants killed by a test set. Mutation score is found by comparing the faults injected to faults found.

Score = (∑ faults found / ∑faults injected) * 100.

For Banking System Application, we injected 61 faults and 49 were revealed from the test cases generated. Using the above formula, we get 80.3% score for Bank system object diagram which shows efficiency level of our approach. It is diagrammatically represented in the form of bar chart as shown in Figure 8 for various operators listed in table 4.

The mutation testing analysis is represented as bar chart in Figure 8.

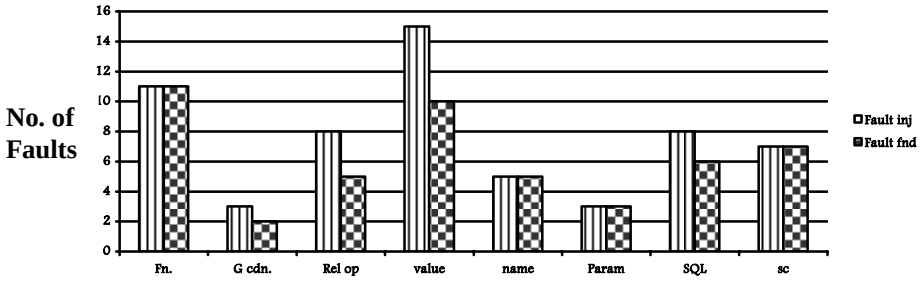

Figure 8: Mutation Testing

We also performed Unit Level and Integration Level Testing and whose results are summarized in Table 5.

Table 5: Experimental Results

Faults	Number of Faults Inserted	Faults found by Aynur, Offutt Approach [10]	Faults found by our approach
Unit Faults	**31**	**24(77%)**	**25(80%)**
Integration Faults	**18**	**15(83%)**	**16(88%)**

6. Conclusion

This paper suggests a model based approach in dealing with object behavioral aspect of the system and deriving test cases based on the Tree structure coupled with Genetic algorithm. Our experimental results shows that it has the capability to reveal **80%** fault in the Unit level and **88%** fault in the integration level. We have viewed testing an application as traversing a path through the DFS for a binary tree to generate appropriate and adequate test cases. The mutation testing conducted has yielded **80.3%** effectiveness in the actual testing process carried out with the generated test cases. Parser and the banking system application have been developed using Java Swing. From the experimental results, we conclude that our methodology is useful to generate test cases after the completion of the design phase and errors could be detected at an early stage in the software development life cycle.

7. Open problem

Our proposed algorithm could be applied for other UML Diagrams like Usecase, Sequence, Collaboration, Activity, State Chart diagrams for generating test cases as a further research in this direction.

8. Acknowledgement

We would like to thank PSG College of Technology Management for providing the necessary facilities to carry out the research work.

References

[1] Bertolino.A, “Software Testing: Guide to the software engineering body of knowledge”, IEEE Software, Vol. 16, 1999, pp. 35-44.

[2] Zhi Quan, Bernhard and Gioranni, “Automated Software Testing and Analysis: Techniques, Practices and Tools”, Proc. of Intl Conf. on System Sciences, HICSS’07, 2007, pp 260.

[3] Emanuela G, Franciso and Patricia,” Test Case Generation by means of UML sequence diagrams and Labeled Transition Systems,” IEEE, 2007, pp.1292-1297.

[4] Monalisa Sarma and Rajib Mall, "Automatic Test Case Generation from UML Models," 10th International Conference on Information Technology, 2007, pp. 196-201.

[5] Philip Samuel, R. Mall, and A.K. Bothra ,"Automatic Test Case Generation Using UML State Diagrams", IET Software, 2008, pp. 79-93.

[6] Iftikhar, "Automatic Code Generation from UML class and State Diagrams", PhD Thesis, University of Tsukuba, Japan, 2005.

[7] Belal and Essam, "The constraints of Object-Oriented Databases", Internation Journal of Open Problems in Computer Science and Mathematics, 2008, Vol.1, No.1,pp. 11-17.

[8] Shaukat Alia, , Lionel C. Briandb, Muhammad Jaffar-ur Rehmana, Hajra Asghara, , Muhammad Zohaib Z. Iqbala, and Aamer Nadeema, "A state-based approach to integration testing based on UML models", Elsevier, Vol. 49, Issue 11 and 12, 2007, pp. 1087-1106.

[9] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan and Juergen Kazmeier, "Automation of GUI testing using a model-driven approach", Proceedings of International workshop on Automation of Software Test, 2006, pp. 9-14.

[10] Aynur Abdurazik, Jeff Offutt and Andrea Baldini, "A Controlled Experimental Evaluation of Test Cases Generated from UML Diagrams", Technical report, George Mason University, ISE-TR-04-03, 2004.

[11] Wang Linzhan, Yuan Jieson, Yu, Hu, LI and Zheng, "Generating Test cases from UML activity diagram based on Graybox method", APSEC, IEEE, 2004, pp.284-291.

[12] Jeff Offutt, Shaoying L and Aynur Abdurazik,"Generating Test Data from State-based Specifications", The Journal of Software Testing, Verification and Reliability, Vol.13, No.1, 2003, pp.25-53.

[13] Dong He Nam, Eric C Mousset and David C Levy, "Automating the Testing of Object Behaviour: A statechart Driven Approach", Proceedings of World Academy of Science, Engineering and Technology, Vol. 11, 2006, pp.145-149.

[14] Niaz I.A and Tanaka J, "An Object-Oriented Approach to Generate Java Code from UML Statecharts", Proceedings of International Journal of Computer and Information Sciences, Vol. 6, 2005.

[15] L. C. Briand and Y. Labiche, "A UML-Based Approach to Application Testing", Proceedings of Journal of Software and Applications Modeling, vol. 1 (1), 2002, pp. 10-42.

[16] A. Cavarra, J. Davies, T. Jeron, L. Mournier, A. Hartman and S. Olvovsky, "Using UML for Automatic Test Generation", Proceedings of ISSTA, 2002.

[17] Peter Frohlich and Johannes Link, “Automated Test Case Generation from Dynamic Models”, ECOOP- Object Oriented Programming, Vol. 1850, 2000, pp. 472-491.

[18] Jeff Offutt and Aynur Abdurazik, "Generating Tests from UML specifications", Second International Conference on the Unified Modeling Language (UML99), 1999, pp. 416-429.

[19] Clay E. Williams, “Software testing and the UML”, International Symposium on Software Reliability Engineering (ISSRE’99), Boca, Raton, 1999.

[20] Mark Priestley, “Practical Object-Oriented Design with UML”, 2nd edition, McGraw –Hill, 2005.