Automatic Test Case Generation for UML Class Diagram using Data Flow Approach

1. M. Prasanna
PSG College of Technology, Coimbatore, India
Email: mp_psg@rediffmail.com

2. K.R. Chandran
PSG College of Technology, Coimbatore, India
Email: chandran_k_r@yahoo.com

3. Devi Bhakta Suberi
PSG College of Technology, Coimbatore, India
Email: dev241282@gmail.com

ABSTRACT

To reduce the high cost of manual software testing and at the same time to increase the reliability of the testing processes, a novel method has been tried to automate the testing process. This paper presents general criteria for generating test cases from UML Class diagram using Data flow testing. The approach of deriving test cases from UML diagrams provides the software tester with a model to generate effective test cases and to commence the testing process early in the software development cycle. The Class diagram of a real time application created using Rational Rose Tool has been taken for generating test cases automatically. The effectiveness of the test cases are evaluated by using a fault injection technique called Mutation Analysis.

Key Words: Software testing, Unified Modeling Language, Use-Pair, Test Cases, State chart Diagram, Class Diagram

1. Introduction

Software testing includes executing a program on a set of test cases and comparing the actual results with the expected results. Testing and test design, as parts of quality assurance, should also focus on fault prevention. 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. A test case is a general software artifact that includes test case input values, expected for the test cases, and any inputs that are necessary to put the software system into a state that is appropriate for the test input values. 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. The basic unit of testing an object-oriented application is a class, and class testing work has mostly centered on functional testing. A state transition diagram can model the dynamic behavior of a single class object if the object has significant event-order behavior. After executing a sequence of methods, the final state that has been achieved by the object can be verified and thus object-oriented classes are well suited to state-based testing. State-based testing mainly examines state change and behavior rather than internal logic, and thus data faults may be missed. Furthermore, data members that do not define an objects state are generally ignored when the classes are validated using state-based testing. Those unexamined data members need to be examined by another technique in order to ensure the quality of the implemented classes. Data flow testing uses the data flow relations in a program to guide the selection of test cases and has been employed to generate data flow test cases for testing object-oriented classes.

Section 2 presents survey work on automated test case generation. Section 3 explains proposed

methodology in the form of flowchart. Section 4 deals with Data Flow approach for Driver Less Train application. Effectiveness of proposed methodology is illustrated in Section 5 and comparison is highlighted in Section 6. Finally Section 7 deals with conclusion and discussion for further research in this area.

2. Related works

Several approaches have been proposed for test case generation. Mainly random, path-oriented, goal-oriented and intelligent approaches. Many researchers have been working in generating optimal test cases based on the specifications. Novada Haji Ali et al [10] have proposed a design of an assessment system for UML diagrams. They developed a tool called UCDA and it generates list of comments on a diagram. Automatic test case generation from UML design diagrams was proposed by Monalisa et al[9]. They transformed UML use case diagram into a graph called Use case diagram graph and sequence diagram into a graph called sequence diagram graph. Based on two coverage criterions, test cases were generated. Emanuela et al[4] have proposed a model for generating test cases from UML sequence diagrams using Labelled Transistion system. Since UML (Unified Modeling Language) is the most widely used language, many researchers are using UML diagrams such as state-chart diagrams, use-case diagrams, sequence diagrams, etc to generate test cases and this has led to Model based test case generation. Optimal test cases can be generated automatically using the proposed algorithm.

3. Proposed methodology

Our proposed methodology involves the following steps:

1. Analyzing the real system which is to be tested and accepted by the user
2. Construct class diagram using rational rose software and store it with ".mdl" as extension.
3. Extract all Data Variables and member functions
4. Select first method to be executed
5. Apply Use-pair method for data variables until Use-pair is not null
6. Finally test sequences are generated

The above steps are illustrated in the form of flowchart as shown in fig 3.1.

4. Case study

In this section, the DriverLessTrain class is used to demonstrate the difficulty of test case generation, infeasible test messages being generated and necessary test messages being missed.

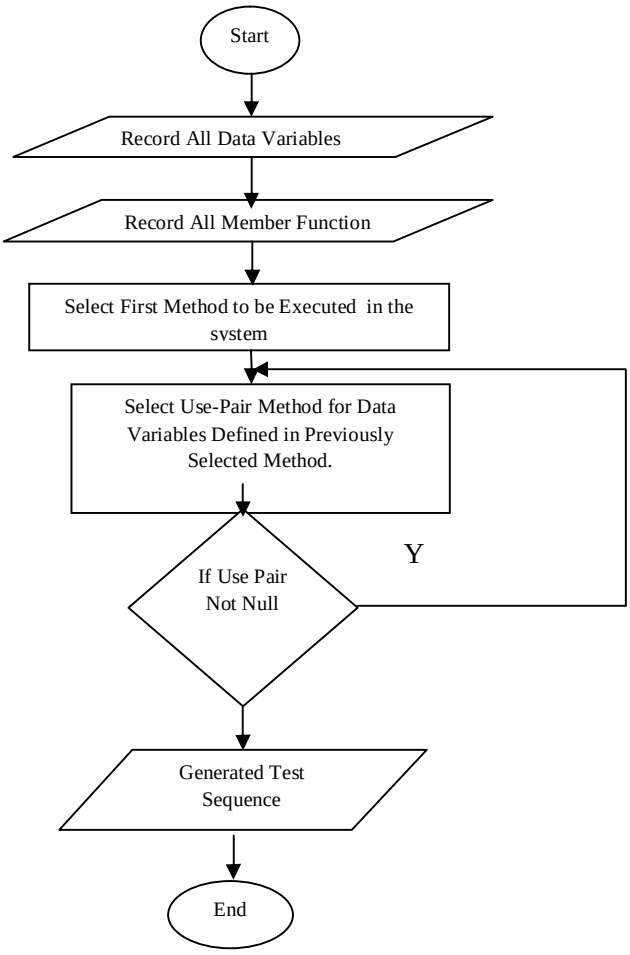


Figure 3.1 Flowchart of proposed Methodology

The data-flow criteria are employed to generate the intra-class data flow test cases. To overcome these weaknesses of generating intra-class data flow test cases, the test cases can be selected from sequences of specification messages. Before the selection, it is necessary to detect if any data flow anomalies occur within the sequences. How to detect intra-class data flow anomalies within the class, to remove the detected anomalies, and to produce feasible intra-class data flow test messages for the DriverLessTrain class are shown in the example.

4.1 Problem statement

The driverless train is an automated train which covers the destination spots without the intervention of humans. Initially the train is in a station where the journey is to be started. At station, Doors are open which allows the entry/exit for passengers. After the passengers get into/out of the train, the doors are closed. The door close event is known by issuing a sound buzzer. A transition called lock doors occurs from door open to door close state. After the doors are closed the train starts with a transition called initiate acceleration. The train is in underway entry, where the traffic control is informed

to monitoring system which constantly monitors the automated train. When the train is in under way entry it maintains a constant speed of 60 kmph. From the underway state the train can take three transitions. From underway if it reaches the stop marker it moves to stop train state which is the final state. From underway if train enters into a station it is considered to be in emergency state.

From the underway station the train may slow down for next station in the process of its journey. For slowing down to a station a transition called pass break marker/apply brake occurs. This is to ensure that the train slows down for station from its constant speed of (60 kmph). After slowing down for a station, the train may reach the stop marker and can enter into the final state. If the journey is not complete, the train then maintains a constant speed of 5 kmph for a specified amount of time and then releases the applied brakes.

On the process, train enters into a station pass slowly through the station where entry is indicated by a sound buzzer and maintains a constant speed of 5 kmph and then reaches the stop marker of that station and slows down and whole process continues until it reaches the final state. When the train has entered into the emergency state, the emergency is cleared or deleted and it reaches the final state.

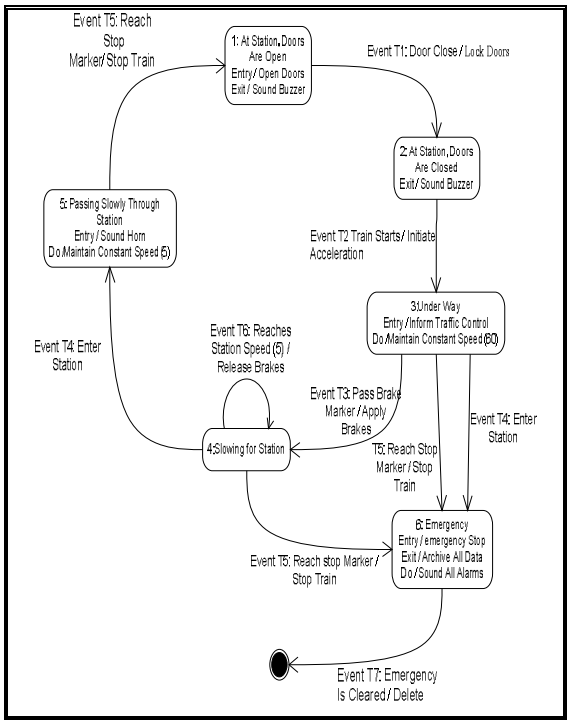


Figure 4.1: State Chart Diagram for Driverless Train

Table 4.1: Transition and their respective edges

Transitions of state chart diagram	Nodes of the graph
Doors close / lock doors	a
Train starts / Initiate acceleration	b
Apply brakes / Pass brake marker.	c
Enter station.	d, e
Reach Stop Marker / Stop train	f, g, h
Reaches station speed (5) / Release brake	I
Emergency cleared / deleted.	J

4.2. Def-use paths of the case study

Data flow testing techniques require directed flow graph that contain the definitions and uses of data variables. These show the data occurrences within programs, and facilitate the computation of define-use pairs. These also help testers to select test cases and to detect whether anomalies exist in the program under test.

To seek the global definitions/uses of data members, only the data occurrences within the public member functions of the class are analyzed. To simplify the define-use presentation of each member function, each code statement is a unit in which the definition and/or use of data members can occur. In Fig 6.2 we see that *is_door_close* data member of *initiate_acc( )* is concerned as to whether it has been properly defined in the preceding functions, i.e *lock_door*(). Take for example the second data member *is_train_start* which has been defined in *initiate_acc*() and used in *apply_break( )*, in which case we understand that only when the train is started, the break can be applied to stop the train. Thus we find here that a data member or a variable defined in one function and used in another function provides us with the combination pair of define-use data flow. This data flow provides us with the information to generate the possible test cases. Similar case is applicable for the following remaining portion of the system.

```
class DriverLessTrain{
        boolean is_door_close;
        //A
        boolean is_train_start;
        //B
        boolean pass_break_marker;
        //C
        boolean is_enter_station_1;
        //D
        boolean is_enter_station_2;
        //E
        boolean reach_stop_marker_1;
        //F
        boolean reach_stop_marker_2;
        //G
```

```
        boolean reach_stop_marker_3;
        //H
        boolean reach_station;
        //I
        boolean emergency_is_cleared;
        //J

        DriverLessT(){ Reset();}

        void Reset(){ is_door_close=false;
        is_train_start=false;
pass_break_marker=false;
        is_enter_station_1=false;
        is_enter_station_2=false;
        reach_stop_marker_1=false;
        reach_stop_marker_2=false;reach_stop_mark
er_3=false;reach_station=false;
        emergency_is_cleared=false;
        }
        void lock_door(){
                if(!reach_stop_marker_3)
        is_door_close=true;
        }
        void initiate_acc(){
                if(is_door_close){
is_train_start=true;}
        }
        void apply_break(){
if(is_train_start){pass_break_marker=true;}
        }
        void enter_station(){
                if(is_train_start)
        is_enter_station_1=true;
                if(pass_break_marker)
is_enter_station_2=true;
                if(reach_station)
        is_enter_station_2=true;
        }
        void stop_train(){

        if(pass_break_marker){reach_stop_marker_2
=true;}
                if(reach_station){
        reach_stop_marker_2=true;}
                if(is_train_start){
        reach_stop_marker_1=true;}

        if(is_enter_station_2){reach_stop_marker_3=
true;}
        }
        void release_break(){

        if(pass_break_marker){reach_station=true;}
        }
        void delete_entry(){
                if(reach_stop_marker_1){
        emergency_is_cleared=true;}
                if(reach_stop_marker_2){
        emergency_is_cleared=true;}
        if(is_enter_station_1){emergency_is_cleared
=true;}
        }
}
```

Figure 4.2: Source code of class DriverLessTrain

4.3 Generating data flow test cases

The test cases generated to cover associations between the definitions and uses of each data member can be yielded from the define-use path. The definitions and uses of data members among the functions of the *DriverLessTrain* class are shown in Table 4.2.

Defined in State	**a**	**b**	**c**	**d**	**e**
Data Members	is_door_close	is_train_start	pass_break_marker	is_enter_station_1	is_enter_station_2
Defined in	lock_door()	initiate_acc()	apply_break()	enter_station()	enter_station()
Used in	initiate_acc()	apply_break() enter_station() stop_train()	enter_station() stop_train() release_break()	delete_entry()	stop_train()

Defined in State	**f**	**g**	**h**	**i**	**j**
Data Members	reach_stop_marker_1	reach_stop_marker_2	reach_stop_marker_3	reach_station	emergency_is_cleared
Defined in	stop_train()	stop_train()	stop_train()	release_break()	delete_entry()
Used in	delete_entry()	delete_entry()	lock_door()	enter_station() stop_train()	NULL

Table 4.2: The definitions and uses of the data members in the member functions of the DriverLessTrain class

The data flow test cases for the given class DriverlessTrain can be generated as *lock_door()* **è** *initiate_acc(),initiate_acc()* **è** *apply_break(), initiate_acc()* **è** *enter_station()* and so on.

Following are the test cases that are generated from Table 4.2 by accounting the define use path.

1. a **è** b **è** c **è** d **è** j
2. a **è** b **è** c **è** e **è** f **è** j
3. a **è** b **è** c **è** e **è** g **è** j
4. a **è** b **è** c **è** e **è** h **è** a

5. a **è** b **è** c **è** f **è** j
6. *a* **è** *b* **è** *c* **è** *g* **è** *j*
7. a **è** b **è** c **è** h **è** a
8. a **è** b **è** c **è** i **è** d **è** j
9. a **è** b **è** c **è** i **è** e **è** f **è** j
10. a **è** b **è** c **è** i **è** e **è** g **è** j
11. *a* **è** *b* **è** *c* **è** *i* **è** *e* **è** *h* **è** *a*
12. a **è** b **è** c **è** i **è** f **è** j
13. *a* **è** *b* **è** *c* **è** *i* **è** *g* **è** *j*
14. a **è** b **è** c **è** i **è** h **è** a
15. *a* **è** *b* **è** *d* **è** *j*
16. a **è** b **è** e **è** f **è** j
17. a **è** b **è** e **è** g **è** j
18. *a* **è** *b* **è** *e* **è** *h* **è** *a*
19. *a* **è** *b* **è** *f* **è** *j*
20. a **è** b **è** g **è** j
21. a **è** b **è** h **è** a

The above sequences have def-use path with respect to the data members but are infeasible sequence except 6, 11, 13, 15, 18 and 19 based on the requirement of the system. The anomaly in the above generated sequence except few of them has resulted due to the presence of more than one different member variables defined in one function and all of those being used in another member function. Thus this results into ambiguity in access of member variables in different functions. The sequence such as *a* **è** *b* **è** *h* **è** *a* which is lock_door() **è** initiate_acc() **è** delete_entry() **è** lock_door() doesn't exist in reality according to the specification of the system. The sequence is an anomaly that must be eliminated from the test case.

4.3.1 Infeasible and ambiguous test cases

Infeasible path problem is the primary practical difficulty in using the all define-use path criterion, as there are many infeasible paths to contend with. In the automatic generation of methods to satisfy the data flow criteria, the problem of generating infeasible sequences is impossible to avoid. A sequence of method calls from outside the class can be specification infeasible or implementation infeasible. Infeasible sequence methods (sub paths) cannot be executed according to the specification. To obtain all possible valid test cases and to reduce the cost of testing, the redundant paths should be removed and the infeasible test cases should be eliminated from the test cases that are generated based on data flow testing criteria.

4.3.2 Selection of feasible test message sequences

If the class under test is implemented by following the state transition diagram, then the paths of transition in the diagram reveal the feasible sequences of member functions of the implemented class. This means the sequences, of member functions (mapping to the paths of transitions) of the object are feasible. Therefore, data flow test cases can be selected from the sequences of member functions based on the def-use pair technique. After traversing the state transition diagram of the *DriverlessTrain* class (Fig. 4.1), the sequences of member functions can be produced as follows.

1. a **è** b **è** c **è** g **è** j
   (*seq. 6 data flow from class*)
2. a **è** b **è** c **è** i **è** e **è** h **è** a
   (*seq. 11 data flow from class*)
3. a **è** b **è** c **è** i **è** g **è** j
   (*seq. 13 data flow from class*)
4. a **è** b **è** d **è** j
   (*seq. 15 data flow from class*)
5. a **è** b **è** c **è** e **è** h **è** a
   (*seq. 18 data flow from class*)
6. a **è** b **è** f **è** j
   (*seq. 19 data flow from class*)

The above sequences can be used as data flow test cases to examine the occurrences of the data member. Here we see that the sequence generated from the state transition diagram is the same as generated from the class *DriverLessTrain*. These sequences are already present in the sequences generated from the class DriverLessTrain. Now we can eliminate those uncommon sequences from the test cases employed among state transition diagram and class *DriverLessTrain*.

5. Mutation analysis

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. The product of mutation analysis is a measure called Mutation Score, which indicates the percentage of mutants killed by a test set.

5.1 Fault Injection

The test cases derived using the define-use path for the Driverless State-Transition diagram is given in Section 4.3. In the fault injection technique, we inject faults into the system by 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.

Table 5.1 Operator and Description

S.No.	OPERATOR	DESCRIPTION
1	Function	Replaces the name of the function
2	Guard	Changes/deletes the guard

	condition	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

For the state transition diagram of the driverless train, we created 43 mutants that use mutation operator discussed above. The summary of the mutants are shown in Table 5.2.

Table 5.2 Summary of the mutants

OPERATOR	FAULTS INJECTED	FAULTS FOUND
Function	4	4
Guard condition	2	2
Relational operator	4	3
Data value	13	9
Data name	5	5
Parameter	6	6
SQL query	3	2
Subclass name	6	6
Total	**43**	**37**

5.2 Mutation Score

Mutation score is found by comparing the faults injected to faults found. For our example, the mutation score is **86%.**

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

The mutation testing analysis is represented as bar chart in Figure 5.1

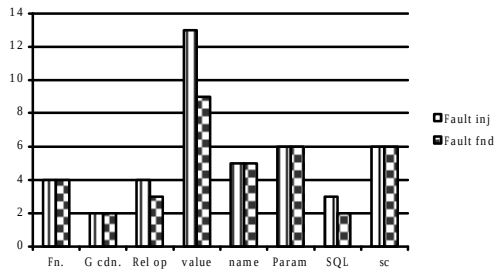


Figure 5.1 Mutation Testing

Total faults injected à 43
Total faults found à 37

6. Comparison

To prove the effectiveness of the test cases generated, we took cruise control problem. Test cases were generated by Random Approach and Data flow approach. 54 test cases were generated by Random methodology but using our proposed approach, only 26 test cases were needed to test the system thoroughly and mutation score for our approach is 92%. It is tabulated as shown in Table 6.1.

Table 6.1 Comparison table

Parameters	Random	Data Flow Approach
No. of test cases	54	26
Faults found	15	23
Faults missed	9	2
Percent coverage	62.5%	**92%**

7. Conclusion

It has been established that UML models can be effectively used to derive test cases. This paper suggests a model based approach in dealing with behavioral aspect of the system and deriving test cases based on Data Flow approach for UML Class diagrams. This approach will help software developer and tester to commence the testing process sufficiently early in the software development cycle. This approach also provides requirements traceability throughout the life cycle, as the models form the basic building blocks of system design. Data Flow approach allow us to use the behavioral information stored in state chart diagrams to generate appropriate and adequate test cases. The generated test case was further considered for the validation. Numerous errors were injected into system and were revealed with the probable occurrence of the each error or fault path in the system. This approach provides efficient fault revealing criteria. Our methodology has been illustrated with a case study of a real world system. We have concentrated on class diagrams and our approach could be extended for Nested State Charts and other UML diagrams for further research in this direction.

References

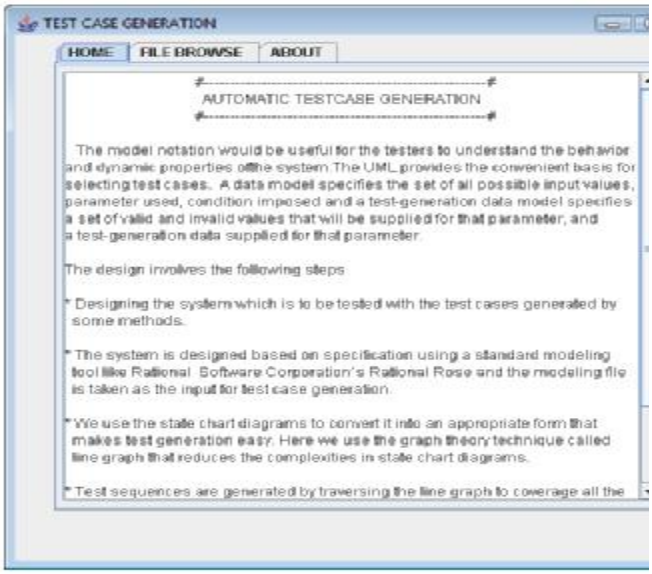
1. Alessandra Cavarra, Charles chrichton, Jim Davies, Alan Hartman, Thierry Jeron and Laurent Mounier, "Using UML for Automatic Test Generation", Oxford University Computing Laboratory, Tools and Algorithms for the Construction and Analysis of Systems, 2000, pp. 235-250.
2. Dong, Eric and David, "Automating the testing of Object Behavior: A Statechart

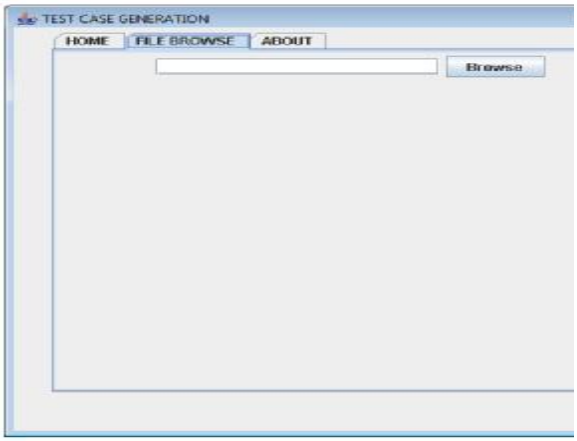
driven approach", WASET, Vol. 11, 2006, pp. 145-149.

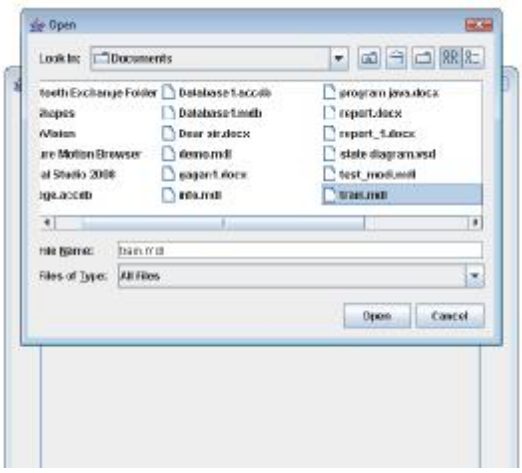
3. ED Adams and Sam Guckenheimer, "Achieving Quality by Design part II using UML", The Rational Edge, IBM, 2004.
4. Emanuela G, Franciso G.O and Patricia D.L, "Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems", Proc. Of IEEE conf. on systems, man and cybernetics, 2007, pp. 1292- 1297.
5. Iftikhar, "Automatic Code Generation from UML class and State Diagrams", PhD Thesis, University of Tsukuba, Japan, 2005.
6. Jeff Offutt, Shaoying Liu, Aynur Abdurazik and Paul Ammann, "Generating Test data from State based Specifications", The Journal of Software Testing Verification and Reliability, Vol.13 (1), 2003, pp.25-53.
7. Mark Blackburn, Aaron Nauman, Bob Busser (Software Productivity Consortium) and Bryan Stensvad, "Defect Identification with Model-Based Test Automation", Society of Automotive Engineers - SAE, Detroit MI, 2003. Vol. 112, No.7, pp. 425-431.
8. Mark Priestley, "Practical Object-Oriented Design with UML", Tata McGraw Hill, 2005, pp. 195-202.
9. Monalisa Sarma and Rajib Mall, "Automatic Test Case Generation from UML Models", 10th International Conference on Information Technology, 2007, pp. 196-201.
10. Novada Haji Ali, Zarina Shukur and Sufian Idris, "A Design of an Assessment System for UML Class Diagram", 5th International Conference on computational Science and Applications, 2007, pp. 539-544.
11. Philip Samuel, R. Mall, and A.K. Bothra ,"Automatic Test Case Generation Using UML State Diagrams", IET Software, 2008, pp. 79-93.
12. 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.
13. Supaporn and Wanchai, "Automated-Generating Test Case Using UML StateChart Diagrams", Proceedings of SAICSIT, 2003, pp. 296 – 300.
14. Wang Linzhang, Yuan Jieson, Yu, Hu, Li and Zheng, "Generating Test Cases from UML Activity Diagram based on Gray-Box Method", APSEC, IEEE, 2004, pp.284-291.

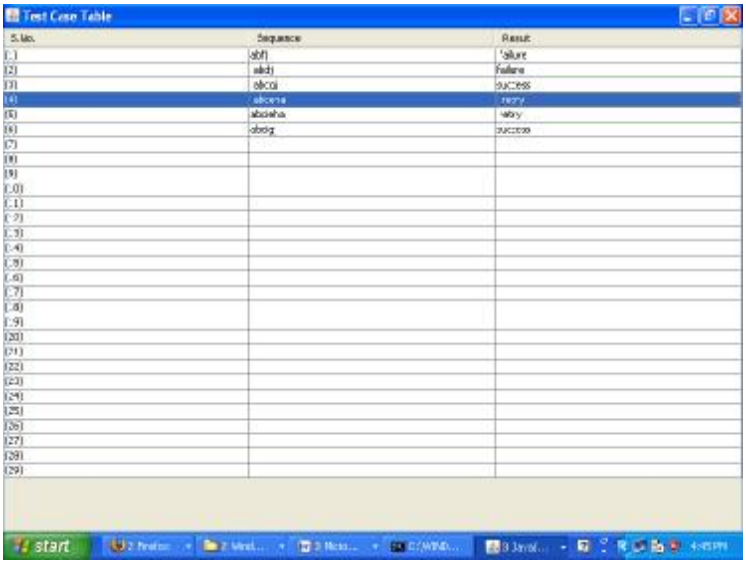
Appendix :

Automatic Test Case generation Tool








Test Case Table
S.No.
Sequence
Result
[1]
abfj
'ailure
[2]
abdj
failure
[3]
abcgi
success
[4]
abcena
nery
[5]
abceha
very
[6]
abcig
success
[7]
[8]
[9]
[10]
[11]
[12]
[13]
[14]
[15]
[16]
[17]
[18]
[19]
[20]
[21]
[22]
[23]
[24]
[25]
[26]
[27]
[28]
[29]
start