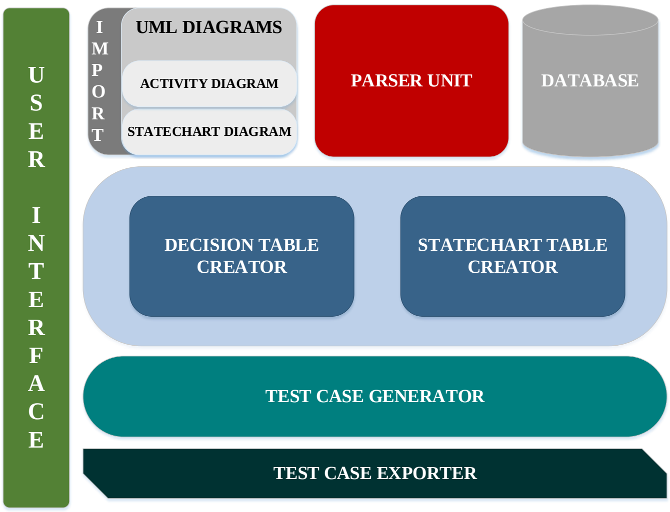

Figure 1: System Design

The modules sketched in the above architecture are delineated as follows:

2.1 Import

In this module we import a model diagram of the application under test. Again we will be focusing only on the activity and statechart diagram created as per the UML 2.0 standards. We assume the diagram to be complete and rich but there is a huge chance for the user to import a UML diagram which may not be in conformity with UML 2.0 standards. In the paper [4] V. Mary Sumalatha and Dr. G. S. V. P. Raju discussed an easy way to remove ambiguities from activity diagrams. This can be applied to make the diagram free from flaws.

2.2 Parser

The diagram imported will be parsed to extract meta-information. In case of an activity diagram we retrieve the swim-lane actors, swim-lane specific actions, decision node, fork node, join node, connectors, dependencies and also its relationship with the previous and next shape. Conversion of any diagram to test cases involves parsing as its initial step. Conversion of statechart diagram to statechart table is simple. With respect to our first approach, we deduced that converting any diagram to a statechart diagram would benefit us rather than forming a separate process for each type of diagrams. Here, the activity diagram shall be converted into a state machine diagram using the following algorithm:

1. Let S0 and S' be the start and exit state of the activity diagram.
2. Let V be the set of all the actions of the activity diagram.
3. Generate new state by applying action from set V to State s0 and store it in set V'.
4. Generate new state by applying next action from set V on last state of set V'.
5. Store the newly obtained connection in a data structure.
6. a) If the obtained shape is a decision box with n outgoing flows then n states are generated and stored in set V'.
   b) If the obtained shape is fork with m outgoing flows then all the m actions are applied on the previous state and new m states are generated.
   c) If the obtained shape is join with m incoming flow then all the m previous states are applied with one action to get one state.
7. Jump to step 3 until exit state is reached and all the actions are covered from set V.

Now with respect to the state chart diagram we extract information such as the states, composite states, submachine states, choice nodes, connectors, dependencies and also its relationship with the previous and next shape.

2.3 Intermediate Form

Defining an intermediate form will ease our efforts by providing a common point to handle data parsed from the different diagrams. Furthermore depending on the approach used we convert the information obtained from the parser to a statechart table or a decision table.

Statechart Table

A statechart table is an alternative way of expressing sequential modal logic. Instead of drawing states and transitions graphically in a statechart diagram, the modal logic is expressed in a tabular format. We will be converting statechart diagram into statechart table because it is a terse, crisp format for a statechart diagram. They also reduce the maintenance of graphical objects. Unlike statechart diagrams, addition and deletion of states into a statechart table will omit the over-head of rearranging states, transitions and junctions.

Statechart Diagram to Statechart Table

This detailed representation not only helps us discover possible transitions of states but also provides us with the adequate information needed to form test cases out of it. The steps included are as follows:

1. Let V be a set of all the states
2. Create a table, where the rows and column are labelled as the states of the system taken in set V. The cell entries are the triggers/actions that cause a transition from a state to another.
3. After implementing the algorithm to generate statechart diagram from activity diagram, information is extracted from the diagram to search for an action between the selected states.
4. If action is found then add the action in the table with the first state as the x-axis and the other as the y-axis

Continue step 4 till end state is reached.

Decision table

As discussed earlier Decision Tables aids ideally to handle transactional situations that represent a table connecting conditions with actions. This tabular representation populates all the conditions in a design model and also checks the actions extracted hence, leading to thorough Test Derivation. Every column in a decision table depicts a test case which brings to notice the Coverage Criteria as at least one test case per combination of conditions is achieved through it. Also adding to its attributes is the Bug Hypothesis which simply is the discovery of improper actions or missing actions that might exist in the model provided [9].

Activity Diagram to Decision Table

After the activity diagram is parsed and relevant shape information is extracted our system will segregate actions based on actors in two classes, user actions and system generated actions. User actions in the swim-lane diagrams very well depict the input conditions to be included in the decision table. Decision nodes also depict conditions on which the application under test will depend. In our approach user actions and decision nodes will act like conditions to the decision table.

The system generated actions depict output actions to be carried out in the decision table. Condition alternatives or otherwise called combinations are generated using cartesian product.This is also called as exhaustive testing.These condition alternatives will be in true/false form. To obtain the value of the expected output we trace the UML Diagram treating it as a tree with the starting node acting as the root. Starting with the start node we traverse the tree to find the output action whose expected outcome is to be calculated .Based on the combination column values we decide the direction of the traversal. Whenever there is an action and a true value is received from the combination for the respective action we move ahead. If a false value is obtained we stop and return a false value to the expected result column. Whenever there is a decision node in the path to the output action the decision of which child to move to depends on the combination value for that decision. Whenever there is a fork node in the path to the output action all the child's of the fork node are traversed one by one to look for the output condition.

Thus after traversing the entire tree based on the above mentioned rules if the output action whose expected outcome we are calculating, is not on the path to the end node we return a false value. On the other hand if find the node on the path to the output action we stop our traversal and return a true value. Thus the rows for expected outcome are calculated and added to the decision table.

2.4 Test Case Generator

Statechart

The statechart table already contains all the information necessary to create a test case. Consider the following e.g.

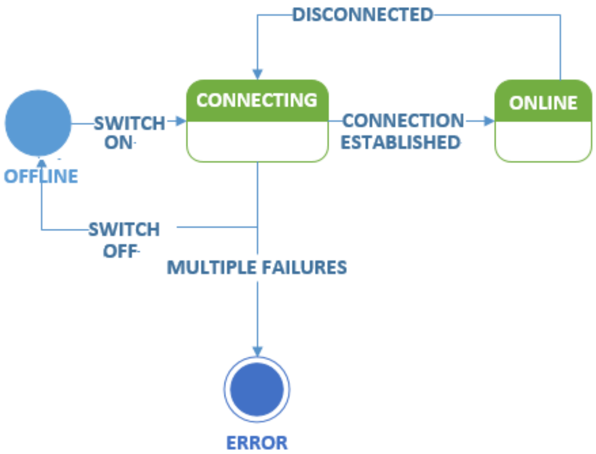

Figure 2: Connection Status

While creating test cases from statechart diagrams care should be taken that all the transitions are exercised at least once. This method of testing ensures optimum coverage without generating large number tests. The statechart table for the diagram is given as below:

States	Offline	Online	Connecting	Error
Offline	-	-	Switch on	-
Online	-	-	Disconnect	-
Connecting	Switch off	Connection Established	-	Multiple Failures
Error	-	-	-	-

: Test Case

Figure 3: Statechart Table

Decision Table

To ease our effort in understanding the procedure of generating test cases through decision table we take up an example of an activity diagram for a file exchanging system.

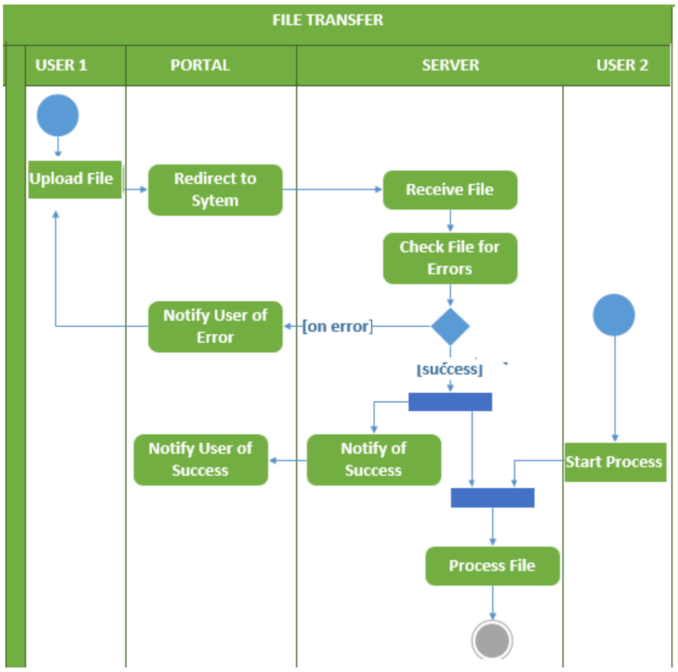

Figure 4: Activity Diagram for a file transfer system

Converting the above system, we obtain the following decision table:

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Upload file	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
On Error	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
Success	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
Start Process	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Actions																
Redirect file to system	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Receive File	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Check file for error	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Notify U for error	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Notify P for success	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Notify U for success	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Process file	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Figure 5: Decision Table

A decision table is capable of providing us test cases as the columns generated in it may act like one. But these test cases may be found as repetitive (i.e. TCs giving same O/P).

To avoid this we may optimize these test cases by collapsing the decision table. If the value of one or more particular conditions can't affect the actions for two or more combinations of conditions, we can collapse the decision table. This can be achieved by keeping the following three steps in mind:

- Combinable columns often but not always next to each other
- Look for two or more columns that result in the same combination of actions (for all the actions in the table)
- Replace the conditions that are different in those columns with "X" (for don't care/doesn't matter/can't happen)

Applying the above with example taken we get [7]

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Upload file	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
On Error	1	1	1	1	0	0	0	0	X	1	1	1	0	0	0	0
Success	1	1	0	0	1	1	0	0	X	1	0	0	1	1	0	0
Start Process	1	0	X	0	1	0	X	0	X	0	1	0	1	0	1	0
Actions																
Redirect file to system	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Receive File	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Check file for error	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Notify U for error	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Notify P for success	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Notify U for success	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Process file	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Figure 6: Collapsing Decision Table

The idea is to repeat this process until no further columns share the same combination of actions It is also important to keep in mind that collapse should not erase an important distinction. In such cases the collapse of two columns is avoided. As a result of which the following collapsed decision table is obtained.

Conditions	1	2	3	4	5	6	7
Upload file	1	1	1	1	1	1	0
On Error	1	1	1	0	0	0	X
Success	1	1	0	1	1	0	X
Start Process	1	0	X	1	0	X	X
Actions							
Redirect file to system	1	1	1	1	1	1	0
Receive File	1	1	1	1	1	1	0
Check file for error	1	1	1	1	1	1	0
Notify U for error	1	1	1	0	0	0	0
Notify P for success	1	1	0	1	1	0	0
Notify U for success	1	1	0	1	1	0	0
Process file	1	0	0	1	0	0	0

Figure 7: Optimized Decision Table

Another aspect to be wary of is a tables that have non-exclusive rules.

2.5 Database

The database operates at two stages in our system. Firstly it is used to store all the meta-data and shape information that will be extracted from the UML diagrams. Secondly the generated test cases will be stored in the database. This will help us to easily export test cases in user customized formats. This information may be useful when our system is to be operated in semi-automatic mode i.e. if the user wants to change some information in the designed model.

3. Acknowledgement

We express our profound gratitude and deep regards to our guides Prof. Kailash Tambe (Assistant Professor, MIT College of Engineering Pune) and Mr. Sunil Deshmukh (Zentest Software Pvt. Ltd., Pune) for their exemplary guidance, constant supervision and encouragement throughout the course of this thesis. This work was supported by Zentest Software Pvt. Ltd., Pune.

4. Conclusion and Future Scope

We have defined a methodology to automatically generate test cases from UML Activity Diagrams and UML State chart Diagrams. We have first parsed these diagrams and converted the parsed information to either decision table or statechart table. We then derive test cases from these intermediate forms. We have discussed two approaches of generating test cases from UML Activity Diagrams. In the future we plan to generate test cases from other UML Diagrams. This will not only help us enhance the quality of testing but also help the tester in deriving test cases which ensures optimum test coverage of the application under test. Now our system takes as input a file containing the design diagram. Presently many applications are used to create UML diagrams for e.g. Rational Rose from IBM, Microsoft's MS Visio, and thus varying file formats. We plan to design a system that will incorporate these different file formats.

We also plan to generate test cases using other black box testing methods. Thus by including boundary value analysis and equivalence class partitioning to our approach we can enrich the automatically generated test cases. Moreover, after obtaining a plethora of test cases we wish to further optimize and prioritize them to obtain an efficient list of test cases. For test case optimization orthogonal array technique can be implemented[8]. OAT is a systematic and statistical way of software testing. Hence OAT will reduce the number of test cases but will retain its coverage. We can also carry out test case prioritization on these improved set of test cases. Clustering approach for test case prioritization is discussed in [6].

References

[1] Model-based Software Testing Ibrahim K. El-Far and James A. Whittaker, Florida Institute of Technology. UML-based Test Generation and Execution.
[2] The Unified Modeling Language User Guide, 2nd Edition Grady Booch, James Rumbaugh, Ivar Jacobson.

[3] http://www.softwaretestinghelp.com/state-transition-testing-technique-for-testing-complex-applications/
[4] V. Mary Sumalatha and Dr. G. S. V. P. Raju, Model Based Test Case Generation from UML Activity Diagrams.
[5] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.
[6] Ryan Carlson, Hyunsook Do, Anne Denton. A clustering approach to improve test case prioritization.
[7] Rex Black, Advanced software test design techniques, decision tables and cause effect graph
[8] Shubra Banerjee, Orthogonal Array Approach for Test Case Optimization
[9] Rex Black, Applying Decision Tables to Business logic.

Author Profile

Melvin Philips is currently pursuing B.E. from Computer Department in Maharashtra Institute of Technology College of Engineering, Pune (MIT-COE) (2010-2014 Batch).

Nikhil Pawar, is currently pursuing B.E. from Computer Department in Maharashtra Institute of Technology College of Engineering, Pune (MIT-COE) (2010-2014 Batch).

Nitesh Joshi is currently pursuing B.E. from Computer Department in Maharashtra Institute of Technology College of Engineering, Pune (MIT-COE) (2010-2014 Batch).

Sanket Khandebharad is currently pursuing B.E. from Computer Department in Maharashtra Institute of Technology College of Engineering, Pune (MIT-COE) (2010-2014 Batch).

Sunil Deshmukh has received training of QAI's CSQA certification. He is currently the Test Lead/Architect at ZenTest Software Pvt. Ltd., Pune, India

Kailash Tambe, has done his M.E. in Information Technology. He is currently an Associate Professor with MIT College of Engineering, Pune.