Automatic Test case Generation From UML State Chart Diagram

Ranjita Swain[1], Vikas Panthi[2]
Rourkela Institute of Mgt. Studies, Rourkela[1]
National Institute of Technology, Rourkela[2]

Prafulla Kumar Behera
Dept. of Comp. Sc.,and Application
Utkal University,
Bhubaneswar

Durga Prasad Mohapatra
Dept. of Comp. Sc. and Engg.,
National Institute of Technology, Rourkela

ABSTRACT
More than 50% of software development effort is spent in testing phase in a typical software development project. Test case design as well as execution consumes a lot of time. So automated generation of test cases is highly required. We present a testing methodology to test object oriented software based on UML state chart diagrams. In our approach we apply function minimization technique and generate test cases automatically from UML state chart diagrams. Here, first the state chart diagram is constructed. Then the diagram is traversed. Here, we perform a DFS to select the associated predicates. After selecting the predicates, we guess an initial dataset. These conditional predicates are, then transformed to generate test cases automatically. Our technique achieves adequate test coverage without unduly increasing the number of test cases. Our approach achieves many important coverage like state coverage, transition coverage, transition pair coverage etc.. This paper also describes how minimization technique is used in testing.

Keywords
Unified Modelling Language, State Chart Diagram, Function Minimization Technique, Test Cases, FSM or EFSM, Model Junit.

1. INTRODUCTION
Testing activities consist of designing test cases that are sequences of inputs, executing the program with test cases, and examining the results produced by this execution. Testing can be carried out earlier in the development process so that the developer will be able to find the inconsistencies and ambiguities in the specification and hence will be able to improve the specification before the program is written [12]. It is still a major problem to meet the requirement specification for the systematic production of high-quality software. Many researchers are doing research on to find effective test cases to minimize time and cost. Hence, it is important to generate test cases based on design specifications [23].

Unified Modeling Language has become the de facto standard for object-oriented modeling and design. It is widely accepted and used by industry [4]. The complexity of system testing can possibly be attributed to the fact that it involves testing a fully integrated system that may be large and complex. Not surprisingly, system testing of typical systems often overwhelms manual test design efforts. Therefore, with continually increasing system sizes, the issue of automatic design of system test cases is assuming prime importance [25]. UML models are popular not only for designing and documenting systems; the importance of UML models in test case design has also been well recognized [25].

The information about a system is distributed across several model views of a system, captured through a large number of diagrams. UML models are intended to help reduce the complexity of a problem, with the increase in product sizes and complexities. Still, the UML models themselves become large and complex involving thousands of interactions across hundreds of objects. Many present day software products are state based. In such systems, the system behaviour is determined by its state. In other words, a system can respond differently to the same event in different states. Therefore, unless a system is made to assume all its possible states and tested, it would not be possible to uncover state-based bugs. Adequate system testing of such software requires satisfactory coverage of system states and transitions. Generation of test specifications to meet these coverage criteria can be accomplished by using the state model of a system. It is a major problem to meet the requirement specification for the Systematic production of high-quality software. However, it is a non-trivial task to manually construct the state model of a system. Therefore, with continually increasing system sizes, the issue of automatic design of system test cases is assuming prime importance [25]. A properly generated test suite may not only locate the errors in a software system, but also help in reducing the high cost associated with software testing.

The UML state model of an actual system is usually extremely complex and comprises of a large number of states and transitions. Possibly for this reason, state models of complete systems are rarely constructed by system developers [25]. In case of component-based software development, test case generation based on program source code proves to be inadequate, where even the source code may not be available to the developers. Hence, it is important to generate test cases based on design specifications [23].

With this motivation, we fix our objective on test case generation, automatically, using UML state chart diagram. The rest of the paper is structured as follows: A brief discussion on UML diagrams is described in the Section 2. Then, we discuss some basic concepts of UML State chart

diagram in Section 3. In Section 4, we explain our methodologies for construction of state chart diagram and test cases generation of using state chart diagram. Section 5 provides the working of our methodology with the SVM (Soft drink Vending Machine) case study. Section 6 explains an implementation of our approach and the experimental studies with result analysis. Section 7 discusses some related work. Finally, Section 8 concludes the paper.

2. UML DIGRAMS

The Unified modelling language is a visual language for specifying, constructing and documenting the artefacts of system [29]. Hence the definition indicates that UML is a language for modelling and representation of systems in general. It reflects various views of a system, in order to capture its different aspects.

UML 2.0 is a whole extensive and more complex than earlier version. The extent of UML documentation has also further increased. The system model consists of four different views each of which emphasizes certain aspects and which are closely related to each other. UML specification defines two major kinds of UML diagram: *structural diagrams and behavioral diagrams.*

*Structural diagram*s show the **static structure** of the system and its parts on different abstraction and implementation **levels** and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Behavioural diagrams show the **dynamic behavior** of the objects in a system, which can be described as a series of changes to the system over **time**.

A state chart diagram specifies the possible states that a model element may assume, the transitions allowed at each state, the events that can cause transitions to occur and the actions that may occur in response to events. Events, states and transitions are the basic components of a state charte diagram. States of an object are essentially determined by the values that certain variables (attributes) of the object may assume. Conceptually, an object continues to remain in a state, until an event causes it to transit to another state. An event is any noteworthy occurrence. An event occurrence may be of some consequence to the system. However, the same event can have different effects (or may even have no effect) in different states. A transition is a relationship between two states indicating a possible change from one state to another. Figure 1 shows the state chart diagram depicting the behaviour of the objects in a simple state chart diagram. A state in a state chart diagram can either be simple or composite type. A simple state does not have any sub-states. A composite state, on the other hand, consists of one or more regions. A composite state can either be sequential or concurrent. A composite state can be in any one of its sub-states, but not in more than one sub-state at any time. On the other hand, in a concurrent type, the state is determined by an object and logic of its sub-states [1]. The object is considered to be in all the concurrent states at the same time.

3. BASIC CONCEPTS

In this section, we discuss some basic concepts which will be used subsequently in our paper. A state chart can be summarized by the following statement:

Usually, a state chart exists in a **current state**. When an **event** occurs, the state chart may take an **action** and may make a **transition** into a **new state**. A range of representations exist for modelling the behaviour of a state chart in the design of the software.

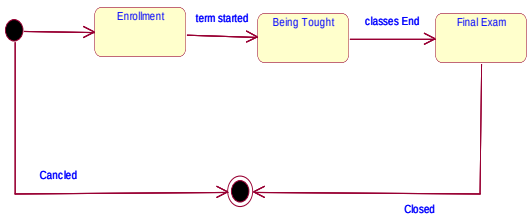

Fig. 1. A Simple state chart diagram Showing States & Events

State : The state of a state chart will normally be represented in software by a **state variable**, represented by a discrete data type. A state chart has a fixed set of possible states. The properties and behavior of the state chart are identical whenever it is in a particular state. A state is an abstraction of the values and links of an object. Sets of values and links are grouped together into a state according to the gross behaviour of the object [25]. For example, the state of a bank could be either solvent or insolvent, depending on whether its assets exceed its liabilities. It is generally represented by a rectangle with rounded corners.

A state may be subdivided into multiple compartments, which are separated from each other by horizontal lines. The different compartments of a state are: Name compartment is optional which holds the name of the state as a string. Internal activities compartment contains a list of internal actions or state activities that are performed while the element is in the state. Internal transitions compartment contains a list of internal transitions. An internal transition executes without exiting or re-entering the state in which it is defined. The Possible states of an object are as follows :

Initial state: A transition leading from an initial event shows the state that an object goes into when it is created or initialized. This is shown as a small black disk. A state chart diagram can have only one initial state.

Final state: Like initial state the state diagram shows final state. It represents the state reached when an object is destroyed, switched off or stops responding to events. This is shown as a small black disk within a large circle. A state chart may have more than one final state.

Activity state: An activity state represents a period of time during which an object is performing some internal processing. As such it is shown as a normal state that contains only an activity. For example, as soon as customer's input to a transaction is complete the activity state becomes active, corresponding to the vending machine working out whether it is capable of returning the change required to complete the transaction.

Event: An event is caused by inputs to a state chart. In response to an event, a state chart may take an action and make a transition to a new state. In any particular state, some events will cause associated transitions to new states, whilst other events will not cause transitions. An event is also an occurrence at a point of time. Events often correspond to verbs in the past tense e.g.(power turned on, alarm set) or to

the honest of some condition e.g.(paper tray becomes empty, temperature becomes lower than freezing) etc.[5].

Action : Associated with a particular state and event may be an action. An action may include a transition to a new state, but may also result in an output from the state chart.

Transitions and Conditions: Transition is an instantaneous change from one state to another.

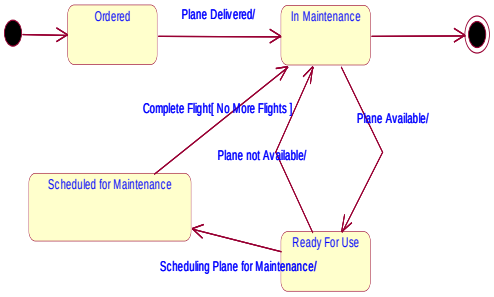

Fig. 2. State chart diagram of Flight Object showing Transitions and Guard Conditions

For example, when a called phone is answered, the phone transitions from the ringing state to the connected state.**:** A transition is triggered by an event occurring in a particular state. In response to the event, a transition is made from one state of a state chart (the **current** state before the transition), to another state of a state chart (the **new** state after a transition).

A guarded transition fires when its event occurs, but only if the guard condition becomes true. A guard condition is checked only once, at the time the event occurs. The transition fires if the condition is true. The UML syntax for a transition is:

event-name argument-list [*guard predicate*]/*action-expression*

4. TEST CASE GENERATION FROM STATE CHART DIAGRAM

In this section, we describe our proposed approach to generate test cases automatically from UML state chart diagrams. First we present few definitions and the relevant test coverage criteria, which will be required in our approach. Then, we describe our approach for the generation of test cases.

4.1 Some Basic definitions

The following terms will be used to describe our methodology.

Def 1 Test case: A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result. A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not [30].

A test case is the triplet [I, D, O], where I is the initial state of the system at which the test data is input, D is the test data input to the system and O is the expected output of the system [18], [19]. The output produced by the execution of the software with a particular test case provides a specification of the actual software behaviour.

Def 2 A State chart graph: It is a diagram, which can be viewed as a graph called a state chart graph G = (N, T), where N is the set of nodes (vertices) of G and T is the set of edges. In G, nodes represent states and edges represent transitions between states. Without any loss of generality, we assume that there is a unique node that corresponds to the initial state and that one or more nodes represent the final states. The initial state is represented as the root of the tree. States at each level of nesting are considered as a sub graph. Sub states of a composite node will become child nodes of that composite node in the tree [23].

Def 3 Sub Path: A sub path P from vertices n_i to n_k is a sequence of nodes n_i, n_{i+1}, . . . , n_k, where for each adjacent pair of nodes (n_{i+j} , n_{i+j+1}) there is an edge in G for 0 <i< k - i.

Def 4 Initial path: Consider a path P on a graph G. A sub path of P that starts from the node representing the initial state is refered to as a initial path of P.

Def 5 Transition path: We consider any sequence of transitions from the initial state to a final state in a state chart graph to be a transition path.

Def 6 Boundary: Every path domain is surrounded by a boarder. A boundary is defined by a set of data points. A boundary consists of several segments and each segment of the boundary is called a border. Each border is determined by a simple predicate in the path condition [9]. In Figure 4, consider the condition

N1(AvailableSoftDrink_Type_1) ≤ 10,

N2(AvailableSoftDrink_Type_2) ≤ 10,

N3 (AvailableSoftDrink_Type_3) ≤ 10 .

Here the variable N represents the number of Available SoftDrink Types requested, in one transaction. The domain of the variable N is the set of all integers. For values of N greater than 10 (N > 10), the condition turns out to be false. A boundary crossing occurs for some input where the conditional predicate changes its Boolean value from true to false or vice versa.

Def 7 Path domain: Consider a path P on a state chart graph. The path condition of the path P is the conjunction of all the individual predicates present along the edges in P. For example, in Fiure 4, $N \leq 10$ and *Returnmoney* ≥ 0 (here *Returnmoney* represents the balance or change remained after calculating the amount) form the path condition for the sub path from *Idle_Machine* state to *SoftDrinkDispenser* state. The conjunction of all the individual predicates present along a initial path is termed as initial path condition. The path domain is the set of all input data values for which the path P is traversed satisfying the path condition (i.e. path condition evaluates to true).

4.2 Coverage criteria

In this section, we discuss some of the relevant coverage criteria which are achieved in our approach.

4.2.1 State Coverage

It covers every state in every state chart for basic test generation. State coverage is a test adequacy criterion that requires tests to check programs' output variables [31]. All

variables still defined when executing in test scope (even those which are not visible, such as private fields of objects) are considered by state coverage.

4.2.2 Transition path coverage

A test set *TS* is said to achieve transition path coverage if given a state chart graph *G*, *TS* causes each possible transition path in *G* to be taken at least once [19]. Cover all arbitrarily long distinct paths through transitions for exhaustive test generation. As there is a defined set of transitions in the state model, a coverage measure associated with this strategy is to measure the proportion of transitions exercised by a set of test cases.

Transition coverage = (Number of transitions exercised)/(Total number of transitions in the state model).

4.2.3 Transition-pair coverage

It is required to cover each pair of adjacent transitions at least once in some test case. Therefore, the transition-pair coverage subsumes the all-transitions coverage. The transition-pair coverage criterion generates more test cases than the transition coverage criterion[33]. For each pair of adjacent transitions S_i : S_j and S_j : S_k in SG, *T* must contain a test that traverses each transition of the pair in sequence [32].

4.2.4 Boundary-testing criterion

The boundary-testing criterion is satisfied for inequality borders. If each selected inequality border *B* is tested by two points (ON-OFF) of test input domain such that, if for one of the point the outcome of a selected predicate *r* is true, then for the other point the outcome of *r* is false. Also the points should satisfy the initial path associated with *B* and the considered points should be as close as possible to each other. We shoud test carefully because domain boundaries are particularly fault prone [11]. Boundary-testing criterion is a criterion for ensuring that a boundary is tested adequately. Instead of generating several test data values that achieve transition path coverage, we only test the border determined by a simple predicate. It helps to reduce the number of test cases significantly; at the same time, the generated test cases achieve very high test Coverage [11].

4.3 AGeTeSC–Our proposed approach to Generate Test Cases

In this section we, discuss our proposed approach to generate test cases from UML state chart diagram. We have named our appoach, Automatically Generating Test cases from State Chart Diagram *(*AGeTeSC*)*.

Our approach for generating test cases is schematically shown in figure 3 . The first step is construct the state chart diagram. The next step is to convert the state chart diagram into state chart graph. Then the graph is traversed to select the predicate functions. In fourth step, we transform the predicate into source code. Then, we construct the Extended Finite State Machine (EFSM) from the code. Finally, we generate the test data corresponding to the transformed predicate functions and store the generated test data for future use. The test case generation steps are discussed below in more detail.

4.3.1 Construction of state chart diagram

First, we construct the state chart diagram. Statecharts offer a system-level view that describes the complete function of a system or application because a statechart diagram captures each possible state of the system. Therefore, the use of statechart helps reduce the possibility of software "hangs" and other unexpected behavior because you are forced to consider every alternative to which the software needs to respond.

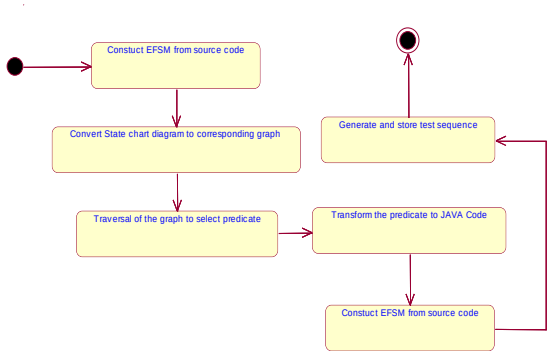

Fig 3 : Figure showing of Test Case Generation scheme

You can design a system so that it scales to handle multiple state reactions and transitions based on any combination of events. Statecharts are similar to graphical dataflow programs in that they are self-documenting and promote the easy transfer of knowledge between developers. A new member of a design team can look at a state chart diagram and quickly grasp the elements of a system.

4.3.2 Conversion of state chart diagram into state chart graph

Then, we convert the state diagram into state graph. a state chart graph G = (N, T), where N is the set of nodes (vertices) of G and T is the set of edges. In G, nodes represent states and edges represent transitions between states. Without any loss of generality, we assume that there is a unique node that corresponds to the initial state and that one or more nodes represent the final states. The initial state is represented as the root of the tree. States at each level of nesting are considered as a sub graph.

4.3.3 Selection of Predicate

Then, we perform a traversal on the state chart graph for selection of predicate. For traversal, we can use any traversal technique like depth first search (DFS) or breadth first search (BFS) to ensure that every transition is considered for predicate selection. In this work, we have used a DFS traversal, as with DFS, it becomes easy to keep track of the initial path in DFS. This also helps in achieving the transition path coverage. All pseudo states are treated at par with a simple state during DFS traversal. For example, within a composite state, the traversal begins with the default initial state or at an entry point. During traversal, conditional predicates are looked, on each of the transition.

4.3.4 Transformation of Predicate into source code

Consider an initial set of data B_0. Here, B_0 consists of all the variables that affect a predicate *r* in the path *P* of a state chart diagram. As mentioned in our approach, we compute two points named ON and OFF for a given border satisfying the

boundary-testing criterion. We transform the relational expressions of the predicates to a function *F* called predicate function. If the predicate *r* is of the form (Exp1 op Exp2), where Exp1 and Exp2 are arithmetic expressions and *op* is a relational operator; then *F* = (Exp1- Exp2) or (Exp2- Exp1) depending on whichever is positive for the data B_0. Next, we successively modify the input data B_0 such that the function *F* decreases and finally turns negative. When *F* turns negative, it corresponds to the alternation of the outcome of the predicate. Hence, as a result of the predicate transformation, the point at which the outcome of a predicate *r* changes, corresponds to the problem of minimization of the function *F*, which is achieved through repeated modification of the input data values. We have transformed these predicate into source code.

4.3.5 Construction of EFSM from the source code

In this step, the Extended Finite State Machine (EFSM) is constructed from the source code automatically. EFSM [7], [24] is very popular for modelling state-based systems like computer communications, telecommunications, and industrial control systems. An EFSM consists of states(including an initial state and an exit state) and transitions between states. A transition is triggered when an event occurs and a condition associated with the transition is satisfied. When a transition is triggered, an action(s) may be performed. Theaction may manipulate variables, read input or produce output.

4.3.6 Generation and Storage of test cases

For finding the minimum of a predicate function *F*, the basic search procedure we use is the alternating variable method [9], [14]. This method is based on minimizing *F* with respect to each input variable in turn. An initial set of inputs can be randomly generated by initializing the data variables. Two data values B_{in} (inside boundary) and B_{out} (outside boundary) are generated using the search procedure mentioned. These two points are on different sides of the boundary. For finding these two data points, a series of moves is made in the same direction determined by the search procedure mentioned above and the value of *F* is computed after each move. The size of the step is doubled after each successful move. This makes the search for the test data quick. A successful move is one where the value computed by the predicate function *F* is reduced. When the minimization function becomes *negative (or zero)*, the required data values B_{in} and B_{out} are noted. These points are refined further to generate a data value, which corresponds to a minimum value of the minimization function along the last processed direction. This refinement is done by reducing the size of the step and comparing the value of *F* with the previous value. Also, the distance between the data points is minimized by reducing the step size. Now, we generate the test data for each conditional predicate in the state chart diagram. Then, the generated test data are stored in a file. Now, we present our *AGeTeSC* algorithm to generate test cases, for Soft Drink Vending Machine , in pseudocode form.

Pseudocode of AGeTeSC algorithm for Soft Drink Vending Machine

Input: State Chart Diagram Diagram, Amt (Money entered), RSDT1(required Soft Drink Type 1), RSDT2 (required Soft Drink Type 2), RSDT3 (required Soft Drink Type 3), {P1, P2, P3(Prices for each softdrink type)}

Output: TS_i(Test Sequence), SC (State Coverage),

TC (Transition Coverage), ACC (Action Coverage), TPC (Transition Pair Coverage), EFSM Graph

IntialState: Start State of Transition

Current State: Current State of The Transition

Final State: Final State of the Transition

```
Begin
State enum {Idle_Machine, Coins_Colletor, Selection_Panel,
Order_Controller, SoftDrinkDispenser,  Change_Dispenser,
Display_For_Customer, Exit}
        If (state=Idle_Machine) then
          Print (TS_i, Current State, Final State)
           State← Selection_Panel
        End if
    If (state= Order_Controller) then
              Print (TS_i, Current State, Final State)
          State ← Selection_Panel
    End if
If (state=Selection_Panel) then
    Print (TS_i, Current State, Final State)
Print(ASDT1,  ASDT2, ASDT3)
state← Display_For_Customer
End if
If (state=Display_For_Customer AND Selectiion=true) then
    Print (TS_i, Current State, Final State)
    Print (RSDT1, RSDT2, RSDT3)
          state← Selection_Panel
End if
If (state=Selection_Panel AND selection=true) then
  If((RSDT1<ASDT1) AND (RSDT2<ASDT2) AND
(RSDT3<ASDT3))
 Print (TS_i, Current State, Final State)
Print(ASDT1 after sell, ASDT2 after sell, ASDT3 after sell)
state← Order_Controller
End if
End if
If (state= Selection_Panel) then
              Print (TS_i, Current State, Final State)
          State ← Money_Collector
    End if
  If (state= Money_Collector) then
              Print (TS_i, Current State, Final State)
          State ← Order_Controller
    End if
    If (state= Order_Controller) then
              Print (TS_i, Current State, Final State)
          State ← Order_Controller
    End if
If (state=Order_Controller AND Selection=true) then
If((RSDT1<ASDT1) AND (RSDT2< ASDT2) AND
(RSDT3<ASDT3))
If(RSDT1>0)
           Total Money= (RSDT1 × MPSDT1)+ (RSDT2×
MPSDT2) +(RSDT3× MPSDT3)
           Return Money= Amt-Total Money
    Print (TS_i, Current State, Final State)
state← SoftDrinkDispenser
if (Amt< Total Money)
Print("Return Money")
state← Idle_Machine
     End if
If(RSDT1 >0)
Print ("Return Money")
```

```
state← Change_Dispenser
End if
If(RSDT2>0)
Print (“Return Money”)
state← Change_Dispenser
End if
If(RSDT3>0)
Print (“Return Money”)
state← Change_Dispenser
End if
End if
If (RSDT1=null AND RSDT2=null AND RSDT3=null)
   Print ( (Should Be ASDT1, ASDT2, ASDT3) > null )
state← Exit
 End if
If (state=SoftDrinkDispenser AND selection=true) then
      Print (TS_i, Current State, Final State)
Print (“Dispense Soft Drink”)
            state←Change_Dispenser
End if
If (state=ChangeDispenser AND selection=true) then
      Print (TS_i, Current State, Final State)
     Print (“Dispense Return Money”)
state← Exit
End if
If (state=Order_Controller AND selection=false) then
      Print (TS_i, Current State, Final State)
state← Exit
End if
End if
If (state=Exit) then
      Print (TS_i, Current State, Final State)
state← Idle_Machine
End if
End
```

5 . WORKING OF THE *AGeTeSC* ALGORITHM

In this section, we explain the working of our *AGeTeSC* algorithm using an SoftDrink Vending Machine (SVM) example as described below.

The SVM (SoftDrink Vending Machine) dipenses softdrinks to the customer on receiving money from them. The state chart diagram of a SVM object for various events of intrest are shown in Figure 4. The object enters into *Idle_Machine* state, when the power is switched on and the different items available on the vending machine are displayed. We have mentioned 3 different categories of drinks. Once the user selects soft drink type in the menu, the object enters into *Display_For_Customer* state and displays pricelist , where the prices of different types of soft drink are displayed. The user can select the type of soft drink needed, as well as the number of softdrinks (N) required. The condition N <= 10 is inserted for the event softdrink selected, as the vending machine is not expected to deliver more than 10 softdrinks of each type in one transaction. Once the type and number of soft drink required are selected, the object enters into *Selection_Panel* state. In this state, the object displays the amount of money (Amt) the user has to insert into the vending machine. Note that

TotalMoney = (N1*P1 +N2*P2 + N3*P3) where N1: No of SoftDrink of Type 1, N2: No of SoftDrink of Type 2, N3: No of SoftDrink of Type 3, P1: Price of SoftDrink Type 1, P2: Price of SoftDrink Type 2, P3: Price of SoftDrink Type 3

As the user enters money (a) the object changes its state to *Order_Controller*. In the *Order_Controller state where* , it calculates how much balance *(ReturnMoney*) is to be returned to the user if any, where ReturnMoney =Amt-TotalMoney.

If the balance is less than zero, the SVM object changes its state from Order_Controller to Change_Dispenser, as the money inserted is insufficient. If the balance is more than or equal to zero, the object goes to *SoftDrinkDispenser* state and delivers the requested number of soft drink. If the balance is zero, then once the soft drink is delivered the machine changes its state from *SoftDrinkDispenser* to idle. If the balance is more than zero, it enters the *Change_Dispenser* state, where the balance money is returned. Once the money is returned, the SVM object transits to *Idle_Machine* state.

From the state chart diagram, we perform a DFS to select the associated predicates. After selecting the predicates, we guess an initial dataset.

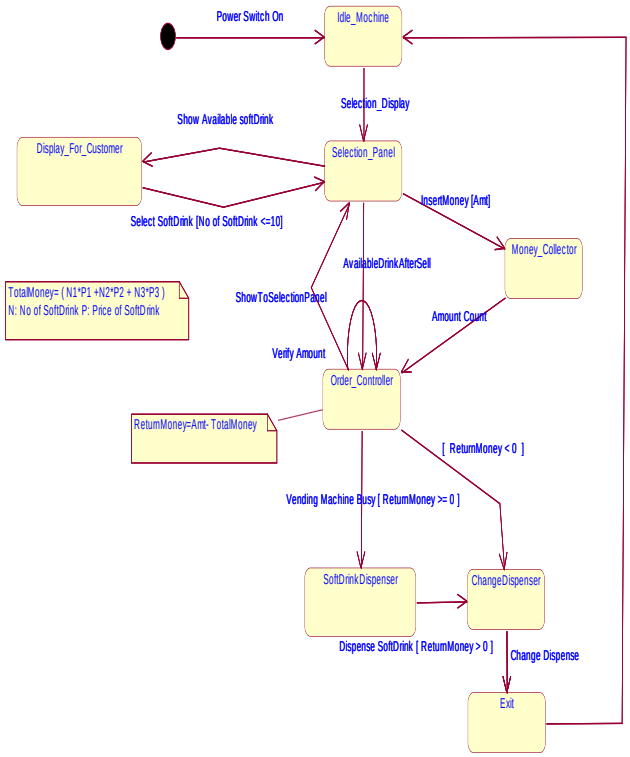

Fig 4 : State Chart Diagram of Soft Drink Vending Machine(SVM)

Consider the boundary is associated with predicate (*ReturnMoney* $\geq$ 0) in the dispenser transition. Let B_0 be the initial data: B_0 = [(5, 5, 5), 300], where (*N1* = N2=N3=5 and *Amt* = 300). Here, it is assumed that the Price for each softdrink type is Rs 15/-. Hence, *TotalMoney* = 5 × *15* + 5 × *15* + 5 × *15* =225 and *ReturnMoney* = 300 − 225 = 75.

The condition ($N \leq 10$) is true for B_0, (as 5 < 10). The function *F* will be represented by the expression *F*: *ReturnMoney* = *Amt* – *totalMoney* i.e *ReturnMoney* = 75. So, $F(B_0)$ = 75. Now, we should minimise *F*, in order to alter the boolean outcome of the predicate (*ReturnMoney* $\geq$ 0), which is true initially. First, we increase the value of *N* in different steps.

1: In the first step, (*N1* = N2=N3=4 and *Amt* = 500), *Total Money*= 4 × *15* + 4 × *15* + 4 × *15* =180 and *Return Money* = 500 − 180 = 320. So, for [*N*, *Amt*] = [4, 500], the function *F* = 320. In the next step, (*N1* = N2=N3=5 and *Amt* = 500), *Total Money*= 5 × *15* + 5 × *15* + 5 × *15* =225 and *Return Money* = 500 − 225 =275. So, for [*N*, *Amt*] = [5, 500], the function *F* = 275. Hence, we observed that *F* decreases with increasing *N*.

2: In the next step, the step size is doubled, i.e the value of *N* is increased by 2. Now, [*N*, *Amt*] = [7, 500]. So, *F* further reduces to *Retunr Money* = 500 −*3*× (7 × 15) = 185.

3: As we double the step size in the next iteration, *n* becomes *N* = *N* + 4 = 7 + 4 = 11, which results in violation of the constraint (*N* ≤ 10) in the softdrink type selected event and number of softdrink in one transaction.

Hence, we reduce the size of the step halved to 2. Now instead of *N* = 11 it becomes *N* = *N* + 2 and we find *N* = 9, Now, [*N*, *Amt*] = [9, 500]. So, *F* further reduces to *Return_Money* = 500 −*3*× (9 × 15) = 95. Now, we reduce the size of the step halved to 1. Hence, N=10 and F=50. But, here as *F*≠ 0, the function is not minimized [23].

4: Then, we select the next variable *Amt* and again decrement / increment operation is carried out to reduce *F*. Here *N* remains constant with 10 and money entered *(Amt*) is reduced in steps.

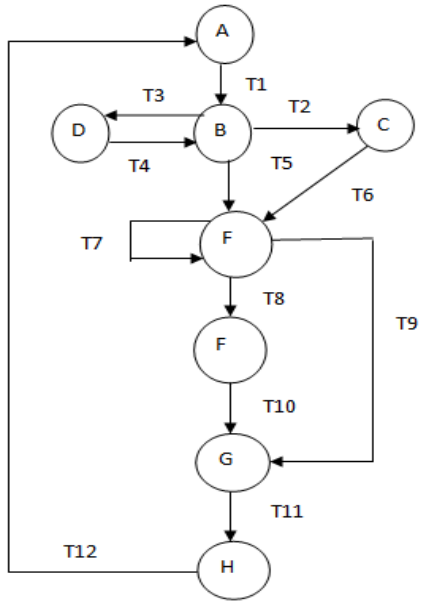

Fig 5. Figure showing the corresponding State Graph from State Chart Diagram

So with [*N*, *Amt*] = [10, 499], *F* becomes *F* = (499 −*3*× (10 × 15)) = 49. Then, we repeatedly reduce *Amt* as [10, 497], [10, 493], [10, 485], [10, 469], [10, 437], by doubling the step size in each iteration. At last *F* = 437 − *3*× (10 × 15) = −13, which is negative for [10, 437].

5: Since *F* has turned negative with step size of 32, we take two initial test data points as B_{in}: [10, 469], and B_{out}: [10, 437].

6: Next, we reduce the step size to half i.e from 32 to 16. So, *Amt* = 469 − 16= 453. Now, with [10, 453], *F* becomes 3 which is positive. And we replace B_{in} as [10, 453] in place of [10, 469]. Again we reduce step size to half i.e from 16 to 8 and with [*N*, *Amt*] = [10, (453 − 8)], we get *F* = 445 −*3* × (10 × 15) = −5. As *F* has turned negative, we replace B_{out} with [10, 445] instead of [10, 437].

7: We replace B_{in} and B_{out} appropriately for reduced values of step size. we get *F* = 0, for the data [10, 450], i.e. *F* = 450 − *3*× (10 × 15) = 0. Finally in this case we obtain the test data as B_{in}: [10, 453] , B_{out}: [10, 445] and *Bout*: [10, 450].

8:The test cases we generate from the predicate (*ReturnMoney* ≥ 0) are :

TC1 = [*Order_Controller*, (10, 453), *dispenser softdrink*, *Return money*],

TC2 = [*Order_Controller*, (10, 450), *dispenser softdrink*]

TC3 = [*Order_Controller*, (10, 445), *Return money*]

where the different fields of the test cases have the format: [input state, test data and expected output state] i.e. [I, D, O].

We require only two test data points B_{in} and B_{out} which are the minimal test points to test the predicate (*Return Money* ≥ 0). Similarly all these generated test values also satisfy all predicates along the transition path from *Idle_Machine* state to *dispenser_softdrink* state during the traversal in the state chart diagram. Hence, these data points correspond to different Boolean outcomes satisfying the initial path condition, which helps us to achieve transition path coverage. We repeat the above procedure to generate test data for each predicate on all transitions in state chart diagram.

6. AN IMPLEMENTATION OF OUR APPROACH

In this section we discuss the results obtained by implementing the proposed approach. The complete approach is implemented using JAVA and Net Beans IDE version 6.0.1. Implementation is done by taking Softdrink Vending Machine as the case study. We have implemented our method for generating test cases automatically from UML state charts in a prototype tool, named Model JUnit. We used Rational Rose to produce the UML design artefact. The architecture of the ModelJunit is shown in Figure 6.

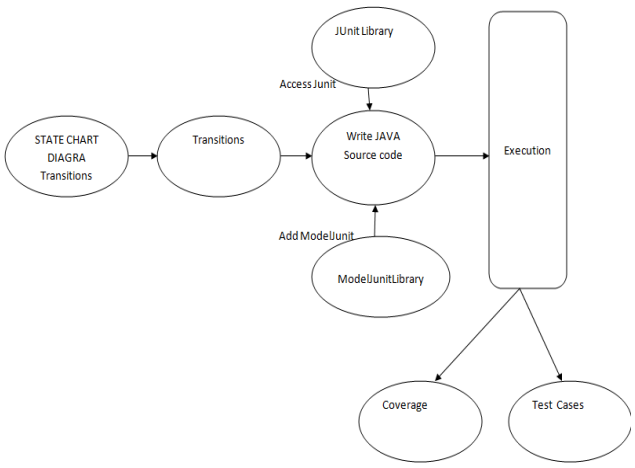

Fig 6. Figure showing architecture of ModelJunit

ModelJUnit is an open source tool, released under the GNU GPL license [34]. ModelJUnit allows us to write simple finite state chart (FSM) models or extended finite state chart (EFSM) models as Java classes, then generate tests from those models and measure various model coverage metrics. Model-based testing allows us to automatically generate test suites from a model of a system under test. ModelJUnit is a Java library that extends JUnit to support model-based testing. ModelJUnit allows us to create simple FSM or EFSM models as Java classes, then generate tests from those models and measure various model coverage metrics. Here, the models are extended finite state charts that are written in a familiar and expressive language: JAVA.

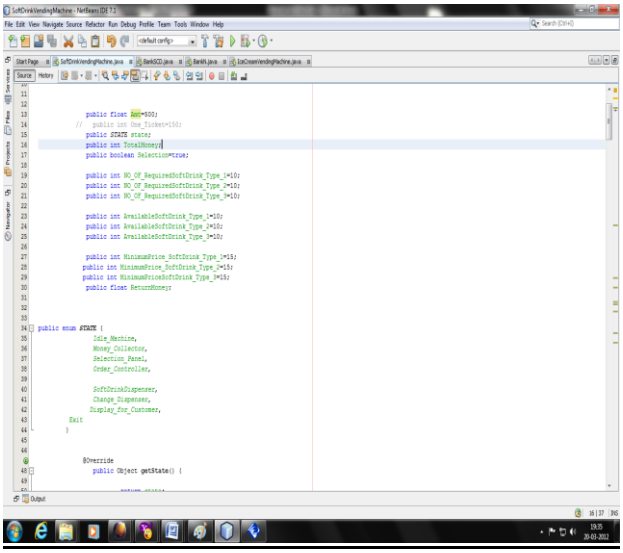

Fig 7 : Screenshot of source code

The source and destination states as well as the prefix path conditions are displayed along with the test data. In our prototype implementation, we have restricted the conditional expressions in state diagrams to have only integer and Boolean variables as these occur commonly. But, other numeric data types can easily be considered. Further, for the prototype implementation we have assumed that the necessary constraints are available in notes. The GUI provides a friendly and efficient user interface to user to generate testing code and connect user defined model with ModelJUnit.

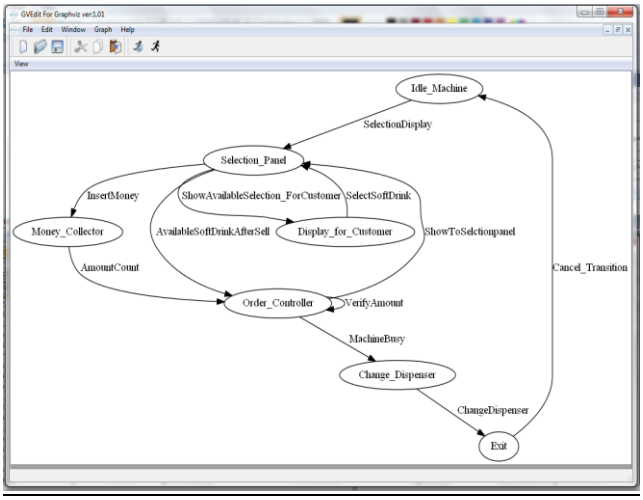

Fig 8 : Screenshot of generated EFSM from source code

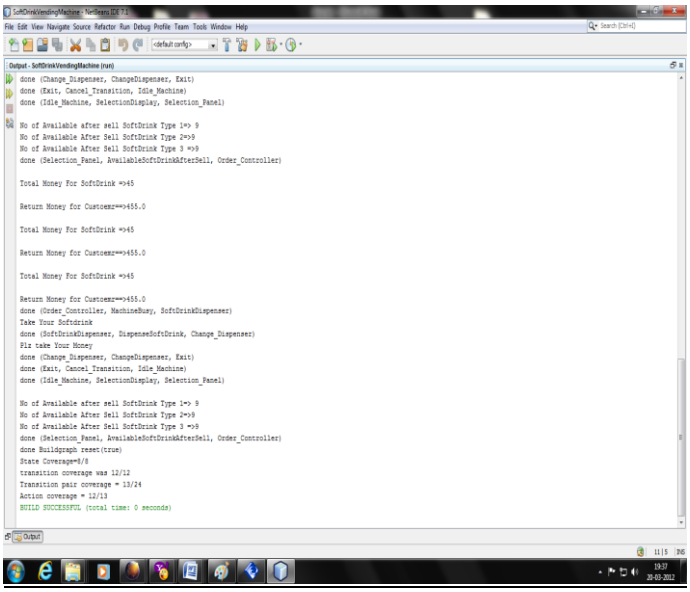

Fig 9 : Screenshot of generated test data with test coverage

TABLE I

TABLE SHOWING TEST COVERAGE ACHIEVED (N:No of softdrink, NS:NO.OF STATES, NT: NO.OF TRANSITIONS, SCP:% OF STATE COVERAGE , TCP:% OF TRANSITION COVERAGE, TPCP: % OF TRANSITION PAIR COVERAGE, AC: % OF ACTION COVERAGE)

Sl. NO	Amt	Selection	N1	N2	N3	ReturnMoney	NS	NT	SCP	TCP	TPCP	AC
1	50	T	1	1	1	5	8	12	100%	100	54.2%	92.3%
2	35	T	4	1	5	Less Money	7	11	100%	100%	56.6%	84.6%
3	45	T	1	1	1	0	8	12	100%	100%	54.2%	92.3%
4	100	F	1	1	3	Not Select item	6	8	100%	100%	69.2%	61.5%
5	500	T	10	10	10	50	6	10	100%	100%	59.09%	76.92%
6	200	T	0	0	0	0 item No select	7	11	100%	100%	56.5%	84.61%
7	500	T	11	1	1	Select More Then Available softdrink	6	10	100%	100%	59.09%	76.92%

The GUI gives the flexibility to view the state diagram. Figure 7 shows the UTG display of the JAVA source file of example mentioned. Figure 8 shows the generated EFSM from the source code. And the set of test cases generated corresponding to our *AGeTeSC* algorithm with the test coverage achieved are shown in Figure 9 . In Figure 9, the initial state, the final state and the test data corresponding to each predicate are also shown. The transition path that is considered while generating the test data in each case is also displayed along with the test data as shown in Figure 9. The percentage of test coverage which are achieved by implementing the case study of SVM object is shown in the Table I.

7. RELATED WORK

In this section, we present some related research work in the area of UML state chart based testing. Generally it is difficult to detect state based faults from the software code. Among all UML diagrams, most probably state chart diagrams have received most attention from researchers to generate test cases [10], [12], [13], [19], [20], [25], [27].

A technique is developed by Offutt and Abdurazik [19], [20] for generating test cases from UML state diagrams. They generated test cases automatically from change events for boolean class attributes. They have highlighted several useful

test coverage criteria for UML state machines such as: (1) full predicate coverage, (2) transition coverage etc. They have derived test cases from state charts focusing on enabled transitions. It appears in [19], [20] that all transitions are assumed to be triggered by change events. Also their [19], [20] approach does not handle guards. In comparison, our approach is not limited to any particular type of event or transition. Our approach can handle change events, time events and transitions with guards.

A method is introduced by Kansomkeat and Rivepiboon [12] for generating test sequences using UML state chart diagrams. They transformed the state chart diagram into a flattened structure of states called testing flow graph (TFG). From the TFG, they listed the possible event sequences which they considered as test sequences. The testing criterion they used to guide the generation of test sequences is the coverage of the states and transitions of TFG.

A method is proposed by Kim et al. [13] for generating test cases for class testing using UML state chart diagrams. They transformed state charts to extended FSMs (EFSMs) to derive test cases. In the resulting EFSMs, the hierarchical and concurrent structure of states are flattened and broadcast communications are eliminated. Then, data flow is identified by transforming the EFSMs into flow graphs, to which conventional data flow analysis techniques are applied.

Also Abdurazik and Offutt [2] proposed test criteria based on collaboration diagrams for static checking and dynamic testing. They adapted traditional data flow coverage criteria in the context of UML collaboration diagrams. It does not generate several test data that achieve transition path coverage, but our approach tests the border determined by a simple predicate, which reduces the number of test cases significantly. Also, our approach achieves transition path coverage. Again our work achieves full predicate coverage as we generate test data for each conditional clause. Again a method is introduced by Korel [14] by using function minimization method in the context of unit testing of procedural programs. He generated test data based on actual execution of the program under test using the function minimization method and dynamic data flow analysis. Test data are developed for the program using actual values of input variables. If during a program execution an undesirable execution flow is observed (e.g. the 'actual' path does not correspond to the selected control path), then the function minimisation search algorithm is used to automatically locate the values of input variables for which the selected path is traversed. In addition, dynamic data flow analysis is used to determine those input variables responsible for undesirable program behaviour, leading to significant speedup of the search process.

Hajnal and Forgacs [9] reported the use of boundary testing that requires the testing of one border (either inside boundary or outside boundary) only along a selected path. They generated two test data points (both inside and outside boundary) of each border of each path. Their testing strategy could also handle compound predicates appearing in a program path. Belli and Holman [3] introduced a coverage-oriented and specification-oriented test approach based on "basic" state charts. They presented a novel representation of statecharts which subsumes common features of different statechart variants. Based on this model and well-defined test criteria, efficient algorithms have been introduced for generating test case sets. Based on this view, test criteria for covering sequences of transitions (k-transition coverage) and faulty transitions (faulty transition pair coverage) were introduced. The test process aims to minimize the total length of test case sets fulfilling these two criteria. Sharma and Mall [25] presented a technique for coverage of elementary transition paths which would also ensure coverage of all states and transitions of the system.

Systa et al. [27] introduced information preserving statechart diagram optimization algorithms for transforming a simple flat statechart diagram into a more compact UML statechart diagram. Their algorithms detected similar responses to certain events and used that information to restructure the diagram. The statechart diagram optimization includes generation of entry actions, exit actions, and actions fired by internal transitions for states. Some actions are also attached to transitions. The conditions under which these actions can be generated for a statechart diagram are characterized.

An elementary set of coverage criteria in software testing is defined by Weiglhofer et al. [28]. Here, test purposes have been presented as a solution to avoid the state space explosion when selecting test cases from formal models. Although such techniques work very well with regard to the speed of the test derivation, they leave the tester with one important task that influences the quality of the overall testing process. Then, they showed how existing tools can be used to efficiently derive test cases and suggest how to use the coverage information to minimize test suites while generating them. It would be straightforward to define further coverage criteria based on logical expressions, such as, multiple condition coverage, or other modified condition/decision coverage variants.

A novel testing technique for object-oriented programs is proposed by Swain et al. [26], which is based on the state and activity models of a system. They have constructed an intermediate representation, named state-activity diagram (SAD) which was used to generate test cases to achieve state-activity coverage of SADs. Their technique could detect seeded integration testing faults.

A methodology is provided by Kosmatov et al. [16] to generate test cases automatically from a given set of test conditions and the input domain. Their approach mainly performs a boundary value analysis on discrete neighborhood of input values and then uses a cost minimization function in the domain to generate test cases automatically.

Also, Gnesi et al. [8] defined a mathematical approach to conformance testing and automatic test case generation from UML state charts. They proposed a formal conformance testing relation for input-enabled transition systems with transitions labeled by input/output-pairs (IOLTSs). Conformance testing is defined as testing the software in order to establish the fulfilment of the specified requirements. A conformance relation defines the correctness criterion of the implementation with respect to the formal specification. IOLTSs provide a suitable semantic model for a behavior represented by a subset of statecharts. They also provided an algorithm which generates a test suite for a given state chart model.

Strategies are made by Offutt and Abdurazik for generating system test cases from state-based formal specifications which have been investigated in [19], [20]. The approach described in [19] is based on designing test cases by transforming statechart diagrams into transition tables, enumerating transition predicates, and then deducing test cases satisfying various coverage levels. In [20], a method to support system level test generation at different coverage levels from state-

based formal specifications has been proposed. The coverage levels include transition coverage, full predicate coverage, transition pair coverage and complete sequence coverage. Their approach assumes that a softwares functionality is described in terms of states and transitions. The approach first derives the transition condition. That is, the condition under which the transition is triggered are determined from the functional specification. These are then represented using algebraic predicates. A specification graph (SG) based on the state-based specification is constructed, where each node represents a state and edges represent possible transitions among states. The SG is then used to derive test specifications for different coverage levels in terms of algebraic predicates. The authors point out that this technique can benefit software developers who construct formal specifications during development [20]. However, construction of formal system level state specifications during development for a practical system is still not very common. Consequently, the applicability of the work reported in their paper is limited.

8. CONCLUSION AND FUTURE WORK

We have defined a methodology to generate test cases from UML state chart diagrams. First, we have constructed the state chart diagram for a given object. Then the state chart diagram is traversed, conditional predicates are selected and these conditional predicates are transformed to source code. Then, the test cases are generated and stored by using function minimization technique.

From the state chart diagram, we perform a DFS to select the associated predicates. After selecting the predicates, we guess an initial dataset. We have generated test predicate conditions from UML state chart diagram, which are used to generate test cases.

Our technique achieves many important coverage like state coverage, transition coverage, transition pair coverage, action coverage. It also achieves full predicate coverage as we generate test data for each conditional clause. It can handle transitions with guards and achieves transition path coverage. Here the number of test cases is minimized and they achieve transition path coverage by testing the boundaries determined by simple predicates. Moreover, our planning to include other diagrams of UML to generate test cases. In future, we will look into how the test cases can be optimized and how other UML diagrams can be combined and used to generate test cases and achieve higher coverage.

9. REFERENCES

[1] *OMG. Unified Modelling Language Specification, version 2.0, Object Management Group, www.omg.org*, August 2005.

[2] Abdurazik, A. and Offutt, J. 2000. *Using UML collaboration diagrams for static checking and test generation, Proceedings of 3rd Int. Conf. UML, Lecture Notes in Computer Science*, 2000, pp.*383 – 395*.

[3] Belli, F. and Hollmann, A. 2008. Test generation and minimization with basic statecharts. In *SAC'08*, March 16-20, 2008.

[4] Binder, R. V. Testing object-oriented software: a survey'. *Software Testing Verification Reliability*, 6(3/4): 1996, pp.125 – 252.

[5] Michel, R. Blaha and James R. Rumbaugh. 2005. *Object-Oriented Modeling and Design with UML*. Pearson, 2nd edition, 2005.

[6] Booch, G., Rumbaugh J., and Jacobson, I. 2001. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2001.

[7] Dssouli, A., Saleh R., Aboulhamid, K., Ennouaary E., and Bourhfir A. 1999. Test development for communication protocols: Towards automation. *Computer Networks*, 31: 1999, pp. 1835–1872.

[8] Gnesi Stefania, Latella, Diego, and Massink Mieke. 2004. *Formal test-case generation for UML statecharts, Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age*, 2004, *pp.75 – 84*.

[9] Hajnal, A. and Forgacs, I. 1998. An applicable test data generation algorithm for domain errors. In *ACM SIGSOFT Software Engineering Notes, Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis*, 1998.

[10] Hartmann, J., Imobedorf C., and Meisinger M. 2000. *UMLbased Integration Testing, Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000.

[11] Jeng, B. and Weyuker, E. J. 1994. A simplified domain-testing strategy. *ACMTrans. Software. Eng. Methodology*, 3(3), 1994, pp.*254 – 270*.

[12] Kansomkeat, S. and Rivepiboon, W. 2003. Automated-generating test case using UML statechart diagrams. In *Proc. SAICSIT 2003, ACM*, 2003, pp.*296 – 300*.

[13] Kim, Y. G., Hong, H. S., Bae D. H., and Cha S. D. et al. 1999. *Test cases generation from UML state diagram, Software Testing Verification and Reliability*, 1999, *pp.187 – 192*.

[14] Korel, B. 1990. Automated software test data generation. *IEEE Trans. Software Engineering*,, 16(8), 1990, pp. *870 – 879*.

[15] Kosindrdecha, N. and Daengdeg, J. 2005. A test generation method based on state diagram. *Journal of Theoritical and Applied Information Technology*, 2005, *pp. 28 – 44*.

[16] Kosmatov, Nikolai , Legeard, Bruno, Peureux, Fabien and Mark, Utting. 2004. Boundary coverage criteria for test generation from formal models. In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering, , Washington, DC, USA*, 2004, pp. *139 – 150*.

[17] Li, H. and Peng, L. C. 2005. Software test data generation using ant colony optimization. In *Proceedings of World Academy of Science, Engineeing and Technology*, January 2005.

[18] Mall, R. 2009. *Fundamentals of Software Engineering*. Prentice Hall, 3rd edition, 2009.

[19] Offutt, J. and Abdurazik, A. 1999. Generating tests from UML specifications. In *Proceedings of 2nd International*

Conference. UML, Lecture Notes in Computer Science, 1999, pp. *416 – 429*.

[20] Offutt, J., Liu, S., Abdurazik, A. and Ammann, P. et al. 2003. Generating test data from state-based specifications. *Software Testing Verification Reliability*, 13, 2003, pp. *25 – 53*.

[21] Pilone, D. and Pitman, N.. *UML 2.0 in a Nutshell*. NY. O'Reilly, USA, 2005.

[22] Priestley, Mark. 2005. *Practical Object-Oriented Design with UML*. Tata McGraw-Hill, 2nd edition, 2005.

[23] Samuel, P., Mall, R., and Bothra, A. K. 2008. *Automatic test case generation using Unified Modeling Language(UML) state diagrams. IET Software, 2(2),* 2008, pp.*79 – 93*.

[24] Savage, P. B., Waiters S. and Stephenson M. 1997. Automated test methodology for operational flight programs. In *Proceedings of IEEE Aerospace Conference*, 1997, pp. 293–305.

[25] Sharma, M. and Mall, R. 2009. Automatic generation of test specifications for coverage of system state transitions. *Information and Software Technology*, (51), 2009, *pp.418 – 432*.

[26] Swain, S. K., Mohapatra, D. P. and Mall, R. 2010. Test case generation based on state and activity models. *Journal of Object Technology*, 9(5), 2010, pp.*1 – 27*.

[27] Systa, T., Koskimiesa, K. and Makine, E. 2002. *Automated compression of state machines using UML statechart diagram. Information and Software Technology, (4),* 2002, pp.*565 – 578*.

[28] Weighhofer, M., Fraser, G. and Wotawa, F. 2009. Using coverage to automate and improve test purpose based testing. *Information and Software Technology*, 51, 2009, pp .*1601-1617*.

[29] UML unified modling language : infrastructure, version 2.0 final adopted specification, September 2003.

[30] IEEE (1998). *IEEE standard for software test documentation*. New York: IEEE. ISBN 0-7381-1443-X.

[31] Koster, K. and Kao, D. C. 2007. State coverage: A structural test adequacy criterion for behavior checking. In ESEC/FSE, 2007.

[32] Offutt, J., Liu, S., A. Abdurazik and P. Ammann. 2003.*Generating test data from state-based specifications.* SOFTWARE TESTING, VERIFICATION AND RELIABILITY *Softw. Test. Verif. Reliab.* 2003; **13**:25–53.

[33] Blanco, R., Fanjul, J. G. and Tuya, J. 2010. *Test case generation for transition-pair coverage using Scatter Search.* International Journal of Software Engineering and Its Applications Vol. 4, No. 4, October 2010.

[34] http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit.