Abstract—In a typical software development project more than 50% of software development effort is spent in testing phase. Test case design as well as execution consumes a lot of time. So automated generation of test cases is highly required. In this paper, we aim at developing and generating test cases from state machine diagram by minimizing cost and time. In our approach, first the state machine diagram is traversed. Then the conditional predicates are selected. These conditional predicates are, then transformed to generate test cases automatically, by applying function minimization technique.

Keywords—Unified Modeling Language; State Machine Diagram; Function Minimization Technique; Test Cases; EFSM; Model Junit.

I. INTRODUCTION

It is a challenging task for analysis of Unified Modelling Language (UML) models since the information about a system is distributed across several model views. UML models are intended to help to 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. It is cumbersome for generating test models like control flow graph from source code. This is especially true of large programs. The UML state-machine diagrams are used for modelling discrete behaviour of an object through finite-state transitions. Such states and transitions are critical to decide the specific operation invocations that would be made based on the conditions arising during a scenario execution. For unit level testing, we can derive tests from UML State machine diagrams, which embody the behavioural description of each component.

Unified Modelling Language (UML) has become the de facto standard for object-oriented modeling and design. It is widely accepted and used by industry [4]. Therefore, UML-based automatic test case generation has turned out to be a practically important and theoretically challenging topic. Not surprising that, it is receiving increasing attention from researchers. UML models are popular not only for designing and documenting systems, but also the importance of UML models in test case design has also been well recognized [25].

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 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]. 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]. 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]. It is obvious that the generation of test cases at an early stage of the software development process, especially design phase, makes coding and testing executed in parallel as well as early detection of a fault [31].

With this motivation, we fix our objective on test case generation, automatically, using UML state machine diagram. The rest of the paper is structured as follows: A brief discussion on UML diagrams is described in the Section 2 and also few UML 2.0 diagrams that are relevant to our test generation methodology are reviewed here. Then, we discuss some basic concepts and definitions of UML State machine diagram in Section 3. In Section 4, we explain our methodologies for construction of state machine diagram and generation of test cases using state machine diagram. Section 5 provides the working of our methodology with the ATM (Automatic Ticket 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.

II. SOME UML 2.0 DIAGRAMS

UML has undergone several revisions since its introduction in the late 90s. UML version 2.0, adds some new capabilities to UML 1.x. In this section, we have restricted our review to those developments which are relevant to our work only.

A. The Three Models

1. Class Model, which represents the static, structural and data aspects of the system. A class diagram expresses the class model.

2. State Model, which represents the temporal, behavioural and control aspects of the system. State machine diagram expresses the state model. It may be noted that the state machine diagram in UML 2.0 was known as state chart diagram in UML 1.x.

3. Interaction Model, which represents collaboration of individual objects and the interaction aspects of the system. Use case, sequence and activity diagram document this model.

A typical software procedure incorporates all the three aspects: It uses data structure (class model), it sequences operations in time (state model), and it passes data and control among objects (interaction model). The three kinds of models separate a system into different views. The different models are not completely independent but a system is more than a collection of these independent parts [5].

Our test generation methodology uses information present in three behavioural diagrams, namely use case, sequence and state machine diagrams.

Sequence diagrams describe how a set of objects interact with each other to achieve a behavioural goal. UML 2.0 introduces the concept of a combined fragment [1] to capture the complex procedural logic in a sequence diagram. A combined fragment is one or more processing sequences grouped together and executed under specific named circumstances. Some of the important fragments are alternatives, option, parallel and loop.

Use case diagrams represent the high level functionalities (called use cases) of a system from the perspective of the users. A use case comprises different possible sequences of interactions between the user and the computer. Each specific sequence of interactions in a use case is called a scenario [5].

A state machine 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 machine 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 machine diagram depicting the behaviour of the objects in a toy train machine. A state in a state machine 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 region is a container for sub-states. The notion of a composite state makes a state machine model a hierarchical diagram. A composite state can either be sequential or concurrent. In a sequential type of composite state, the state is considered to be an exclusive-or of its sub-states. That is, 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.

![Fig. 1. State machine diagram of a Toy train headlight](image)

...
III. BASIC CONCEPTS AND DEFINITIONS

A state machine diagram is a graph, which consists of nodes and directed arcs. The directed arcs interconnect the nodes. Here, the nodes represent states and directed arcs represent transitions. The state machine diagram captures the different states that an object undergoes and models the dynamic behaviour [5]. In UML, the behavioural specification of an object is represented by a state machine diagram for the object. A state machine specifies an object’s responses to the events it might detect during its lifetime.

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. Similarly, the states of a telephone could be ringing, dialling or waiting. Each object can only be in one state at a time and passes through a number of states in its life time. A state satisfies some conditions on its variables, performs some action or waits for some event. 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 as follows:

Name compartment: This optional compartment holds the name of the state as a string.

Internal activities compartment: This compartment contains a list of internal actions or state activities that are performed while the element is in the state.

Internal transitions compartment: This compartment contains a list of internal transitions. An internal transition executes without exiting or re-entering the state in which it is defined.

Fig. 2. The initial, final and composite states of a CD player

Nested States: A state can either be simple or composite. A simple state is a state that does not have sub states(states within a state) [23]. The composite state consists of different nested sub states as given in Figure 2. The sub states are grouped together in the enclosing state because they share common properties.

This new state exists only to group together as shown in Figure 2 and Figure 4, related states and does not introduce any new behavioural possibilities. Composite states have the following properties:

1. If a composite state is active, exactly one of its sub states must also be active.
2. An event that is detected while an object is in composite state can trigger outgoing transitions from either the composite state itself or from the currently sub state.
3. The nested states within a composite state form a sort of sub-state machine. Like the normal states, a composite state can contain initial and final states.

Fig. 3. State machine diagram of a CD player showing the history state of a CD Player

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 machine diagram can have only one initial state.

Final state: Like initial state the state diagram also 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 machine 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, in Figure 4 as soon as customer’s input to a transaction is complete the activity state becomes active, corresponding to the machine working out whether it is capable of returning the change required to complete the transaction.

Historic State: It causes an implicit transition to the last active sub state, as required. History states are represented by the uppercase alphabet H, as shown in Figure 3 within a circle and can only appear inside composite states. A transition to a history state causes the sub state that was most recently active in the composite state to become active again [22].

Event: An event is 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].
Events include error conditions as well as normal occurrences, e.g. motor jammed, transaction aborted etc. There are several kinds of events. The most common are the signal event, change event and the time event.

**Signal event:** A signal event is an event of sending or receiving a signal. Usually we are more concerned about the receipt of signal, because it causes effects in the receiving objects. A signal is a message between objects while a signal event is an occurrence in time.

**Change event:** A change event is an event that is caused by the satisfaction of a Boolean expression. The intent of a change event is that an expression is continually tested, whenever the expression changes false to true, the event happens [5]. The UML notation for a change event is the keyword *when* followed by a parenthesized Boolean expression. Some examples of change events are:

- when (room temp < heating set point)
- when (room temp > cooling set point)
- when (battery power < lower limit)
- when (tire pressure < minimum pressure)

**Time event:** It is an event caused by the occurrence of an absolute time or the elapse of a time interval. The UML notation for an absolute time is the keyword *when* followed by a parenthesized expression involving time. Similarly, for an elapse of a time interval the keyword *after* is used.

- when (date = January 1, 2012)
- after (25 seconds)

**Transitions and Conditions:** Transition is an instantaneous change from one state to another. For example, when a called phone is answered, the phone transitions from the ringing state to the connected state. The transition is said to fire upon the change from the source state to the target state. A transition fires when its event occurs. The general format of the label of a transition is as follows:

\[
\text{event[guard-constraint]} \Rightarrow \text{activity-expression/action}
\]

A guard condition is a Boolean expression that must be true in order to occur a transition. For example, in Figure 5 a traffic light at an intersection may change only if cars are waiting on the road. 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. A guard condition is different from a change event. A guard condition is checked only once while a change event is, in effect, checked continuously.

**IV. GENERATING TEST CASES USING STATE MACHINE DIAGRAM**

In this section, we describe our proposed approach to automatically generate test cases from UML state machine 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.

**A. Some Basic definitions**

The following terms will be used to describe our methodology.

**Definition 1: Test case:** A test case is the triplet [I, S, O], where I is the initial state of the system at which the test data is input, S 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.

**Definition 2: Levels of testing:** In object-oriented systems, testing is generally done at three different levels of abstraction: class level, cluster level and system level. Class-level testing tests the code for each operation supported by a class as well as all possible method interactions within the class. Class level testing also requires testing all methods in each of the states that a corresponding object may assume. At cluster-level testing, the interactions among cooperating classes are tested. System-level testing is carried out on all the clusters making up a complete system.

**Definition 3: State machine graph:** A state machine diagram can be viewed as a graph called a state machine graph.
that each clause in the predicate is tested independently by a test suite. In other words, a test suite is said to achieve full predicate coverage if each clause in each predicate on every transition is made to independently affect the outcome of the predicate. Given a test set TS and state machine graph G, TS must cause each clause in every predicate on each transition in G to take on the values TRUE and FALSE in turn. But all the other clauses in the predicate have values, such that the value of the predicate will be the same as the clause being tested [23]. This ensures that each clause in a condition is separately tested.

3) 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 should 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]

C. STUPEC–Our proposed approach to Generate Test Cases

In this section we, discuss our proposed approach to generate test cases from UML state machine diagram. We have named our approach, State based Testing Using PrEdicate Conditions (STUPEC).

Our approach for generating test cases consists of 3 steps.

• Step 1: Selection of Predicate.

• Step 2: Transformation of the predicate.

• Step 3: Generation of Test cases.

1) Selection of Predicate: First, we construct the state machine diagram. Then, we perform a traversal on the state machine 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.

2) Transformation of Predicate: Consider an initial set of data Bo. Here, Bo consists of all the variables that affect a predicate r in the path P of a state machine 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 $(\text{Exp1 op Exp2})$, where Exp1 and Exp2 are arithmetic expressions and op is a relational operator; then $F = (\text{Exp1-Exp2})$ or $(\text{Exp2- Exp1})$ 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.

3) Generation 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 initialising 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 minimisation 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 minimisation 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 minimised by reducing the step size. Now, we generate the test data for each conditional predicate in the state machine diagram. Then, the generated test data are stored in a file.

Now, we present our STUPEC algorithm to generate test cases, in pseudocode form.

**Pseudocode of STUPEC algorithm:**

**Input:** State Machine Diagram, $n$(number of persons travelling in one ticket) and $(1 \leq n \leq 6)$, $amt$(Amount of money entered to machine).

**Output:** $B_{in}$ (In boundary value), $B_{out}$ (Out boundary value), $TS$ (A test suite).

$OPENQueue$: A queue which contains nodes which are to be processed.

$CLOSEQueue$: A queue which contains nodes which are already processed.

$C$: Number of consecutive iterations in the loop.

$F$: Predicate Function to be calculated.

$p, pn$: intermediate variables used for calculation.

**Begin**

Read trainfare // Here trainfare is the ticket price for one person which is supplied as 150
Read $n$ // $1 \leq n \leq 6$, Here we have supplied $n = 2$
$C = 2$

$CF = NF = 0$ // Two variables $CF$ (Current predicate function ) and $NF$ (Next predicate function) are considered to calculate predicate function

$TS = NULL$

Read $amt$ // Here we have supplied $amt = 1000$
$totalfare \leftarrow n \times$ trainfare

$OPENqueue$ from $Initialstate$

while $OPENqueue \neq$ EMPTY

For each transition Do

Begin

For each path in the state machine graph Do

Begin

while ($n \leq 6$)
$\text{totalfare} \leftarrow n \times \text{trainfare}$
$NF \leftarrow \text{amt} - \text{totalfare}$
if $n = 6$ then
if $NF = 0$ then
Print “The current state is issue ticket state”
end if
Print “The function is not minimized”
end if
if $n > 6$ then
Print “Violation Of Constraint”
end if
if $NF < CF$ then
$n \leftarrow n+2$
else
$n \leftarrow n+1$
end if
$CF = NF$

For each transition Do

Begin

while ($n \leq 6$)
$\text{totalfare} \leftarrow n \times \text{trainfare}$
$NF \leftarrow \text{amt} - \text{totalfare}$
if $n = 6$ then
if $NF = 0$ then
Print “The current state is issue ticket state”
end if
Print “The function is not minimized”
end if
if $n > 6$ then
Print “Violation Of Constraint”
end if
if $NF < CF$ then
$n \leftarrow n+2$
else
$n \leftarrow n+1$
end if
$CF = NF$

Now, we present our STUPEC algorithm to generate test cases, in pseudocode form.
V. WORKING OF THE STUPEC ALGORITHM

In this section, we explain the working of our STUPEC algorithm using an automatic ticket machine (ATM) example as described below.

The ATM (Automatic Ticket Machine) issues travelling tickets to passengers on receiving money from them. The state machine diagram of an ATM object for various events of interest are shown in Figure 6. The object enters into idle state, when the power switch is on. Once the user selects ticket type button in the menu, the object enters into the display destination state, where the ticket fare to different destinations are displayed. The user can select the destination, ticket type and the number of persons (n) to travel. The condition n ≤ 6 is inserted for the event ticket type selected, as we have assumed that the ticket machine is not expected to issue a ticket for more than 6 persons in one transaction, as happens in Indian Railway Reservation system. Once the ticket type and number of persons required are selected, the object enters into the display money state. In this state, the object displays the amount of money (totalfare) the user has to insert into the ticket machine. Note that here totalfare = ticket fare × (n). As the user inserts money (amt) into the machine, the ATM object changes its state from Display money state to busy. In the busy state, it calculates how much balance or change (chng) has to be returned to the user, if any, where chng = (amt − totalfare). If the change balance is less than zero, the ATM object changes its state from busy to return money as the money inserted is insufficient and returns the inserted amount. If the change balance is more than or equal to zero, the ATM object goes to issue ticket state and issues the ticket for requested number of persons. If the balance is zero, then once the ticket is issued, the machine object changes its state from issue ticket to idle. If the balance (chng) is more than zero, the ATM object enters into the return money state, where the change balance money is returned. Once the money is returned, the ATM object again enters into idle state.

From the state machine diagram, we perform a DFS to select the associated predicates. After selecting the predicates, we guess an initial dataset. Consider the boundary is associated with predicate (chng ≥ 0) in the issue transition. Let B₀ be the initial data: B₀ = [2, 1000], where (n = 2 and amt = 1000). Here, it is assumed that the train fare for a particular destination for each person is Rs.150. Hence, totalfare = 2 × 150 = 300 and chng = 1000 − 300 = 700.

The condition (n ≤ 6) is true for B₀, (as 2 < 6). The function F will be represented by the expression F: chng = amt − totalfare i.e. chng = 700. So, F(B₀) = 700. Now, we should minimise F, in order to alter the Boolean outcome of the predicate (chng ≥ 0), which is true initially. First, we increase the value of n in different steps.

Step 1: In the first step, n = 3, totalfare = 3 × 150 = 450 and chng = 1000 − 450 = 550. So, for [n, amt] = [3, 1000], the function F = 550. Hence, we observed that F decreases with increasing n.

Step 2: In the next step, the step size is doubled, i.e. the value of n is increased by 2. Now, [n, amt] = [5, 1000]. So, F further reduces to chng = 1000 − (5 × 150) = 250.

Step 3: As we double the step size in the next iteration, n becomes n = n + 4 = 5 + 4 = 9, which results in violation of the
constraint \((n \leq 6)\) in the ticket type selected event and number of persons in one transaction.

Hence, we reduce the size of the step halved to 2. Now instead of \(n = 9\) it becomes \(n = n + 2\) and we find \(n = 7\), which again results in violation of the constraint as \(n > 6\). Again we reduce the step size to 1 and \(n\) becomes 6 with \(F = 1000 - (6 \times 150)\). But, here as \(F \neq 0\), the function is not minimized [23].

**Step 4:** Then, we select the next variable \(amt\) and again decrement / increment operation is carried out to reduce \(F\). Here \(n\) remains constant with 6 and money entered \((amt)\) is reduced in steps.

So with \([n, amt] = [6, 999]\), \(F\) becomes \(F = (999 - (6 \times 150)) = 99\). Then, we repeatedly reduce \(amt\) as \([6, 997], [6, 993], [6, 985], [6, 969], [6, 937], [6, 873]\), by doubling the step size in each iteration.

At last \(F = 873 - (6 \times 150) = -27\), which is negative for \([6, 873]\).

**Step 5:** Since \(F\) has turned negative with step size of 64, we take two initial test data points as \(\text{B_no: [6, 937]}\) and \(\text{B_out: [6, 873]}\).

**Step 6:** Next, we reduce the step size to half i.e. from 64 to 32. So, \(\text{amt} = 937 - 32 = 905\). Now, with \([6, 905]\), \(F\) becomes 5 which is positive. And we replace \(B_{\text{no}}\) as \([6, 905]\) in place of \([6, 937]\). Again we reduce step size to half i.e. from 32 to 16 and with \([n, amt] = [6, 905 - 16]\), we get \(F = 889 - (6 \times 150) = -11\). As \(F\) has turned negative, we replace \(B_{\text{out}}\) with \([6, 889]\) instead of \([6, 873]\).

**Step 7:** We replace \(B_{\text{no}}\) and \(B_{\text{out}}\) appropriately for reduced values of step size, we get \(F = 0\), for the data \([6, 900]\), i.e. \(F = 900 - (6 \times 150) = 0\). Finally in this case we obtain the test data as \(\text{B_no: [6, 905]}, \text{B_out: [6, 900]}\) and \(\text{B_out: [6, 889]}\).

**Step 8:** The test cases we generate from the predicate \((\text{chg} \geq 0)\) are:

- \(\text{TC1} = [\text{busy}, (6, 905)], \text{issue ticket}, \text{return money}\),
- \(\text{TC2} = [\text{busy}, (6, 889)], \text{return money}\)
- \(\text{TC3} = [\text{busy}, (6, 900)], \text{issue ticket}\)

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

We need only two test data points \(B_{\text{no}}\) and \(B_{\text{out}}\) which are the minimal test points to test the predicate \((\text{chg} \geq 0)\). Similarly all these generated test values also satisfy all predicates along the transition path from \(\text{idle state to issue ticket state}\) during the traversal in the state machine 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 machine diagram.

**VI. AN IMPLEMENTATION OF OUR APPROACH**

This section discusses 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 Automatic Ticket Machine as the case study.

![Fig. 6. State machine diagram for Automated Ticket Machine](image)

We have implemented our method for generating test cases automatically from UML state machines in a prototype tool, named ModelJUnit. We used Rational Rose to produce the UML design artefact. The architecture of the ModelJUnit is shown in Figure 7.

![Fig. 7. Architecture of Our Proposed Method](image)

Model-based testing allows you 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 you to write simple FSM or EFSM models as Java classes, then generate tests from those models and measure various model coverage metrics. Models are extended finite state machines that are written in a familiar and expressive language: JAVA. ModelJUnit is an open source tool, released under the GNU GPL license. ModelJUnit allows you to write simple finite state machine (FSM) models or extended finite state machine (EFSM) models as Java classes, then generate tests from those models and measure various model coverage metrics. ModelJUnit is an open, available test case generation tool.
tool, using an extended finite state machine diagram[29]. ModelJUnit searches the graph of an EFSM at runtime.

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. The action may manipulate variables, read input or produce output. The condition associated with a transition is a Boolean predicate that must evaluate to TRUE in order for the transition to be taken. EFSM models are graphically represented as graphs where states are represented as nodes and transitions as directed edges between states.

An extended finite state machine (EFSM) is a 7-tuple \(< S, C, I, O, T, s_0, c_0 >\) where \(S\) is a non-empty set of main states, \(C = \text{dom}(v_1) \times ... \times \text{dom}(v_l)\) a non empty countable set of contexts with \(v_i \in V\), \(V\) the non-empty finite set of variables and \(\text{dom}(v_i)\) a non-empty countable set referred to as the domain of \(v_i\), \(I\) a non-empty finite set of inputs, \(O\) a nonempty set of outputs, \(T \in S \times C \times I \times O \times S \times C\) the set of transition relations, \(s_0 \in S\) the initial main state, and \(c_0 \in C\) the initial context of the EFSM.

<table>
<thead>
<tr>
<th>No</th>
<th>S, Tranfer,Amt</th>
<th>NS</th>
<th>NT</th>
<th>SCP</th>
<th>TCP</th>
<th>TPCP</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6, 150, 900</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>54.54</td>
</tr>
<tr>
<td>2</td>
<td>6, 150, 800</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>54.54</td>
</tr>
<tr>
<td>3</td>
<td>7, 150, 800</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>45.45</td>
</tr>
<tr>
<td>4</td>
<td>3, 150, 500</td>
<td>7</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>63.63</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4, 150, 100</td>
<td>7</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>63.63</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6, 150, 700</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>54.54</td>
</tr>
<tr>
<td>7</td>
<td>3, 150, 800</td>
<td>7</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>63.63</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7, 150, 600</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>45.45</td>
</tr>
<tr>
<td>9</td>
<td>2, 150, 100</td>
<td>7</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>63.63</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9, 150, 700</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>54.54</td>
</tr>
<tr>
<td>11</td>
<td>3, 50, 1500</td>
<td>7</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>63.63</td>
<td></td>
</tr>
</tbody>
</table>

TABLE I

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 [30]. The test designer needs to design states and actions based the behaviour of system under test (SUT) to make a model. The GUI is developed using the swing component of Java. A GUI screen along with a sample state diagram is shown in Figure 8. The GUI gives the flexibility to view the state diagram. Figure 9 shows the UTG display of the JAVA source file of example mentioned and the set of test cases generated corresponding to our STUPEC algorithm.

It also shows the test coverage achieved. The percentage of test coverage which are achieved by implementing the case study of ATM is shown in the Table I.

In this figure, the initial state, the final state and the test data corresponding to each predicate are 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.

VII. RELATED WORK

State machine diagrams were known as state chart diagrams in UML 1.x version. 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].

Offutt and Abdurazik [19], [20] developed a technique 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.

Kansomkeat and Rivepiboon [12] have introduced a method 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.

Kim et al [13] proposed a method 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.

Abdurazik and Offutt [2] also 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.

Korel [14] introduced a method of 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] presented a coverage-oriented and specification-oriented test approach based on “basic” state charts. They introduced a novel representation of state charts which subsumes common features of different state chart 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 state chart diagram optimization algorithms for transforming a simple flat state chart diagram into a more compact UML state chart diagram. Their algorithms detected similar responses to certain events and used that information to restructure the diagram. The state chart 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 state chart diagram are characterized.

Weighlofer et al [28] defined an elementary set of coverage criteria in software testing. 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.

Swain et al [26] proposed a novel testing technique for object-oriented programs, 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.
Kosmatov et al [16] provided a methodology 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 neighbourhood of input values and then uses a cost minimization function in the domain to generate test cases automatically.

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 labelled 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 behaviour represented by a subset of state charts. They also provided an algorithm which generates a test suite for a given state chart model.

Offutt and Abdurazik made strategies 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 state chart 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 software’s 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.

VIII. CONCLUSION AND FUTURE WORK

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

In this work, from the state machine diagram, we performed a DFS to select the associated predicates. After selecting the predicates, we guessed an initial dataset. We generated test predicate conditions from UML state machine diagram, which are used to generate test cases.

Our approach 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 that achieve transition path coverage by testing the borders determined by simple predicates. Moreover, we plan 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 used to achieve higher test coverage.

AUTHORS PROFILE

Ranjita Swain completed her MCA from College of Engg. And Technology, OUAT, Bhubaneswar, India. She is pursuing her Ph.D degree from Utkal University, Vani vihar, Bhubaneswar, India. She is currently working as Senior Lecturer in Computer Science Dept., Rourkela Institute of Management Studies, Rourkela, India. She has 11 years of teaching experience and her fields of interest are Software Engg., Discrete, Mathematical Structure.

Vikas Panthi completed his B.E at Samrat Ashok Technological Institute, Vidisha, India. He is a M.Tech student at computer science department in National Institute of Technology, Rourkela, India. His special fields of interest include Software Engineering, DBMS.

Prafulla Ku. Behera has received his Ph.D degree from Utkal University, Vani vihar, Bhubaneswar, India. He is currently working as a reader at Dept. of Computer Science & Application, in Utkal University, Vani vihar, Bhubaneswar, India. His special fields of interest include Mobile Computing, Software Engineering, He is a member of CSI.

Durga Prasad Mohapatra received his Ph. D. from Indian Institute of Technology Kharagpur and M. E. from Regional Engineering College (now NIT), Rourkela. He joined the faculty of the Department of Computer Science and Engineering at the National Institute of Technology, Rourkela in 1996, where he is now Associate Professor. His research interests include software engineering, real-time systems, discrete mathematics and distributed computing and published more than forty papers in these fields. He has received many awards including Young Scientist Award for the year 2006 by Orissa Bigyan Academy, Prof. K. Arumugam award for innovative research for the year 2009 and Maharashatra State National Award for outstanding research for the year 2010 by ISTE, NewDelhi. He has also received three research projects from DST and UGC. Currently, he is a member of IEEE. Dr. Mohapatra has co-authored the book *Elements of Discrete Mathematics: A computer Oriented Approach* published by Tata McGraw Hill.
REFERENCES


