Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Transition coverage based test case generation from state chart diagram

Sonali Pradhan [a,*], Mitrabinda Ray [a], Santosh Kumar Swain [b]

[a] Department of Computer Science and Engineering, SOA University, India
[b] School of Computer Engineering, Kalinga Institute of Industrial Technology (KIIT), India

ARTICLE INFO

Article history:
Received 19 October 2018
Revised 24 April 2019
Accepted 15 May 2019
Available online 16 May 2019

Keywords:
Software testing
UML state chart diagram
Coverage criteria
State-based faults
Test case
Test suite

ABSTRACT

State-based testing is a challenging area in the software testing field. This paper proposes a set of algorithms to generate test cases from a state chart diagram based on various coverage criteria. The objective is to find various types of state-based faults by covering states and transitions of an object. First, the state chart diagram is transformed into an intermediate graph, State Chart Intermediate Graph (SCIG). Then, for a given coverage criteria, traversing method is applied on SCIG to generate test cases. Different trees from SCIG based on various coverage criteria are extracted for test case generation. Various coverage criteria such as All Transition (AT), Round Trip Path (RTP) and All Transition Pair (ATP) are considered. We introduce algorithms for two most efficient state-based criteria, RTP and ATP. Two case studies, *Stack Operation and Vending Machine Automation system,* are discussed throughout the paper. We experimentally observed that (i) AT consumes the most test resources (ii) ATP can't achieve 100% transition coverage (iii) test cases generated based on RTP is efficient, and it overcomes the transition explosion problem of AT. This analysis is beneficial in the area of semi-automatic test case generation in model-based testing.

1. Introduction

In Software Development Life Cycle (SDLC), software testing has a significant role in the software development process (Sharma and Misra, 2017; Aljawarneh et al., 2017; Pressman, 2005; Mall, 2018; Fitzgerald and Stol, 2017). The intention is to find out bugs in the program (Lonetti and Marchetti, 2018; Matalonga et al., 2017; Misra et al., 2018). Earlier tester went for manual testing and that was a prolonged and tedious process which required more time to execute. Then the tester went for automated testing in the software development process. Till today it has been a challenge to have fully automated software (Mathur, 2013). There are two types of software testing named as Black box testing and White box testing. Black box testing is also known as functional testing (Bohme et al., 2017; Zhou et al., 2018; Nidhra and Dondeti, 2012). In white box testing or code-based testing, the requirement of the source code is necessary and early fault detection in the specification or design phase is not possible (Neto et al., 2008; Shafique and Labiche, 2010; Mostowski, 2019; Da Silva, 2015). Model-Based Testing (MBT) has an essential role in finding fault earlier than code-based testing (Bohme et al., 2017; Pradhan et al., 2019; Utting et al., 2012; Utting et al., 2016).

For object-oriented modelling and design, Unified Modeling Language (UML) has now become the commonly accepted standard (Alhir, 2002; Bauer and Odell, 2005; Burgueno et al., 2018). The behaviour of the state model can be achieved through MBT (Salman and Hashim, 2016; Swain et al., 2012a). The UML state chart diagram is one of the behavioural diagrams, which is used to generate test cases in state-based testing (Hashim et al., 2018; Pradhan et al., 2019). The goal is to achieve the state-based faults which cannot be detected from the source code. Those state-based faults can only be detected in state-based testing.

With this motivation, we fix our objective to detect maximum state-based faults at the design level. For this, we set various coverage criteria in state-based testing and propose algorithms for each coverage criteria. In our approach, test paths are generated using various coverage criteria like AT, RTP, ATP, ATP with length2 (LN2), ATP with LN3 and ATP with LN4, where coverage analysis is considered for the percentage of state and transition coverage. We compare the efficiency of various coverage criteria in terms of number of test cases required and fault detection rate.

* Corresponding author.
E-mail addresses: sonalipradhan7@gmail.com (S. Pradhan), mitrabindaray@soauniversity.ac.in (M. Ray).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

The development of the paper is structured as follows: Section 2 provides a background study. Section 3 discusses the related work on state-based testing. Section 4 describes the proposed algorithms for various coverage criteria in state-based testing. Section 5 elaborates the analysis and effectiveness of various state-based coverage criteria. Finally, Section 6 concludes the paper and suggests future work.

2. Background study

In this section, we discuss the concept of State-Based Testing (SBT) which includes a state chart diagram of our case study Stack Operation and state-based faults. SBT is deriving test cases from the system and examining the dynamic behaviour of the model. State chart diagram is also known as a state diagram where different state coverage, transition coverage, and event coverage occur. Fig. 1 shows a state chart diagram of Stack Operation (Pradhan et al., 2019) as our case study.

In Fig. 1, there are five states Initial, Empty, Holding, Full and Final. The state coverage, event coverage and transition coverage are tested from the state chart diagram. For example, let the capacity of the stack is three, after two add events with the condition if (stack size < max size), it holds in the *Holding* state. Moreover, at the third push (add a particular element as an event), the stack goes with the state change from *Holding* state to *Full* state with the condition if (stack size = max size) and then *Full* to *Final* state.

The state-based faults are identified at the time of detecting a valid and invalid transition (Antoniol et al., 2002; Kabir, 2017). Some significant state-based faults are given below with all associate definitions (Ray and Mohapatra, 2012).

Definition 1 (*Incorrect transition*). *Message $m_k$is invoked on object o_l. It will trigger changes to object o_m where the transition moves to $s_x \rightarrow s_y$, instead of $s_x \rightarrow s_z$.*

Definition 2 (*Missing transition*). *Message m_k is invoked on object o_l, the object o_l changes to object o_m without triggering any changes to o_m.*

Definition 3 (*Incorrect state*). *In the state s_x, a message m_k is to be sent from an object o_l to state s_y. Instead the above process in state s_y, in state s_x, a message m_k is sent from an object o_m, when the state is s_y.*

Definition 4 (*Unspecified/corrupt state*). *From the object o_l message m_k is invoked, where it changes to object $o_m$and instead of the state change s_x(of object o_m) $\rightarrow s_y$ (of object o_l), it changes to s_z and state s_z is not specified in the design.*

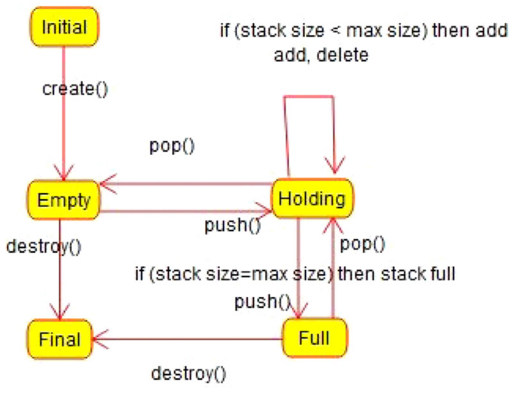


Fig. 1. State diagram of Stack Operation.

Definition 5 (*Missing action*). *A message m_k is sent to object o_m, which in turn, it changes the state without producing any action. The state change will be s_x of object o_m.*

Definition 6 (*Incorrect action*). *A message $m_k$is sent to object o_m, which in turn should change the state $s_x$of object o_m, but the state changes to s_y.*

Definition 7 (*Unspecified events*). *Without any specified event, a state s_x of object $o_l$is changed to state s_y of object o_m, when a message is invoked to object o_l.*

3. Related work

In Offutt's state-based technique (Offutt et al., 2003), a test data generation method has been defined by Abdurazik and Offutt (1999), where test requirements must cover during testing, and the description of the test cases are the test specifications. The test data generation method is used for the test case generation using a state chart diagram. They proposed TSL language and verified some elements as values verification, precondition values, exit command and check the expected output. The elements are derived from preconditions in the state chart diagram and with the triggering events. It makes the automation better but, it is challenging to consider the non-Boolean type of variables in automation and to analyse the dynamic behaviour of the system, Prasanna and Chandran (2009) considered internal and external effects of the system. This process is capable of detecting 80% of faults and 88% faults in unit testing and integration testing respectively. They also went for mutation analysis to measure the effectiveness of the test cases where results come to 80.3% in MBT. In their approach, they generate test cases using the genetic algorithm; it is proved to be a better approach to generate effective test cases. After exploring the approach of test automation, cluster level state-dependent behaviour has been explored.

As reported by Kansomkeat and Rivepiboon (2003), the testing process partially solves by automatically generated test cases from UML state chart diagrams. First, the diagram is transformed into an intermediate graph, called Testing Flow Graph (TFG). Then the test cases are generated from the intermediate graph using various testing coverage criteria. The mutation analysis is done to assess the fault revealing power of the test cases. The limitation is that the testing technique solves the testing process partially. Functional minimisation technique is used by Swain et al. (2012b) to generate test cases. Test cases are derived using state coverage, transition pair coverage, and action coverage. In that work, it has been observed that test cases are generated by minimising the cost and time. The future challenge is to use this technique in the concurrent states and using other UML diagrams to generate test cases.

In a study as proposed by Utting et al. (2006), LEIRIOS Test Generator (LTG) model-based testing tool is embedded with MBT approach to generate test cases and executable test scripts using various coverage criteria. LTG supports various coverage criteria to generate test cases. This approach deploys in some critical domains. LTG tool is the combination of OCL notations and pre/post notations. This combination supports both UML state chart machine and abstract machine. Hence, it is beneficial by using these two notations as OCL and pre/post. OCL is a part of UML, and it performs an important role in the analysis phase of the software development process.

The cost-effectiveness of the test cases has been analysed by Briand et al. (2004) using the coverage criteria AT, ATP, Transition Tree (TT) and Full Predicate (FP). They found that AT coverage criteria is not sufficient to detect faults. The ATP is the most reliable coverage criteria (with no certainty) but expensive (cost is 3 to 7 times more than AT). They find that TT is not always cost-effective. On the other hand, FP coverage criteria appeared to be not very cost-effective but effective than ATP.

An industrial case study is used to examine the cost-effectiveness using state-based testing (Holt et al., 2014). In that study, a MBT tool is composed and used to generate automated test cases, 26 real faults have been collected to evaluate the test suites. The results show that the detailed test model and rigorous testing of the system under test increases the fault detection ability of the SUT. The cost-effectiveness of the SBT gives a combined effect of coverage criteria, test model, test oracle and sneak paths (testing unspecified behaviour). In this paper, coverage analysis is done using coverage criteria AT, ATP, and Round Trip Path (RTP).

Recent research is going on towards the applications of search-based techniques (SBTs) for MBT. Experimental analysis on SBTs for MBT is presented, and the limitations are discussed (Saeed et al., 2016). They addressed both functional and structural coverage of the model-based testing. Many research gaps are found for the existing applications such as multi-objective search-based testing, applying constraints handling, applying the hybrid technique, addressing data, requirement adequacy criteria, addressing visualisation of the landscape.

An assessment of the most common type of Extended Finite State Machines (EFSMs) test selection criteria are presented in work El-Fakih et al. (2017). They take the test selection criteria as the test suite that covers single and double transfer faults, single output parameter faults, and different types of single assignment faults in an EFSM specification design. The model has been transformed into an intermediate graph, and the graph is traversed using some state-transition criteria. This paper describes test selection criteria over selected EFSM faults. They find random test suites perform well in comparison to other test derivation strategies.

4. Proposed test case generation framework

Our proposed test case generation framework outlines in this section. A flowchart appears in Fig. 2, which shows the entire procedure to generate test cases.

The steps for generating test cases in the flowchart are as follows:

Construct the state chart diagram of the system using the *Star-UML* tool
Convert the state chart diagram into an intermediate graph.
Apply various algorithms on the intermediate graph to generate all possible test paths.
Generate test cases from the testing paths of the coverage criteria.

4.1. Conversion of the state chart diagram into an intermediate graph

Two typical cases studies, *Stack Operation* (Pradhan et al., 2019) and *Vending Machine Automation systems* (Swain et al., 2012c) are considered throughout the paper. These two case studies are moderate size popular application, and there are a limited number of objects associated with this and the objects are modal. It is easy to explain our approach throughout the paper with these case studies. From the state chart diagram, the XMI code is generated using starUML. In Stack Operation, the number of transitions and states are 8 and 5 respectively. The snapshot of the Java-based program using J-Unit tool in Fig. 3 is used to show the states and state change of Stack Operation. An intermediate graph, State Chart Intermediate Graph (SCIG) is generated from the state chart

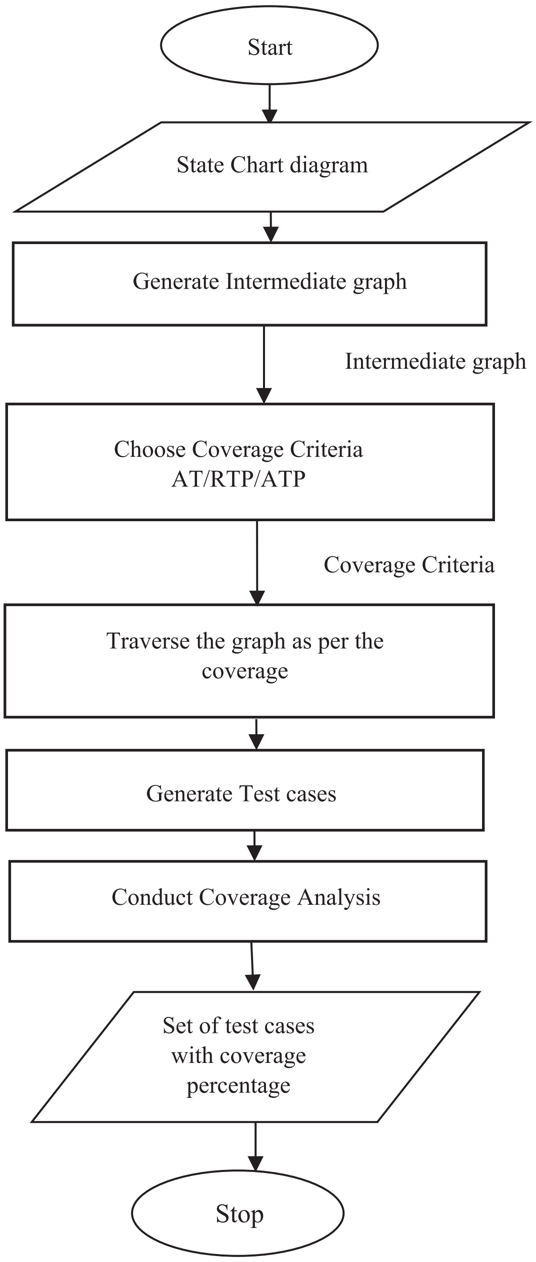
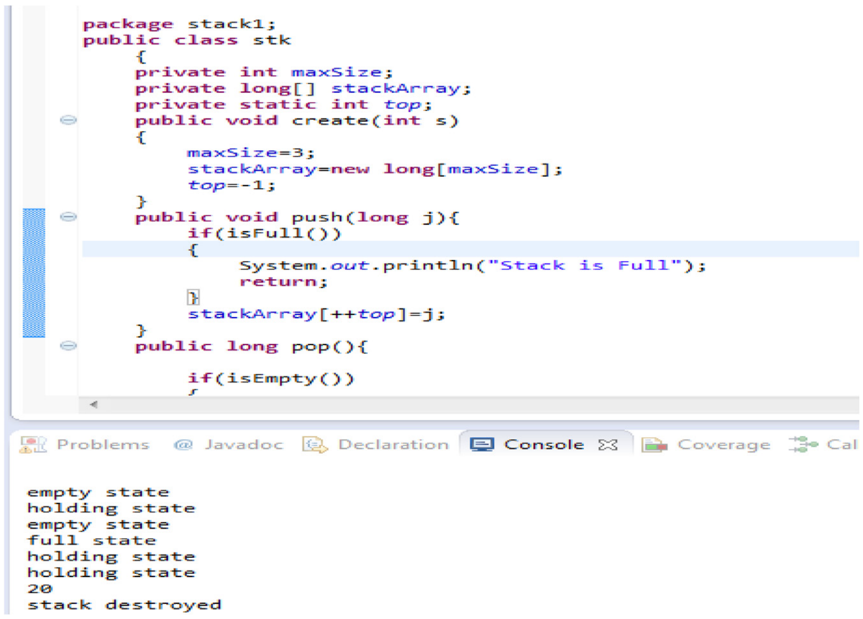

Start
State Chart diagram
Generate Intermediate graph
Intermediate graph
Choose Coverage Criteria
AT/RTP/ATP
Coverage Criteria
Traverse the graph as per the coverage
Generate Test cases
Conduct Coverage Analysis
Set of test cases with coverage percentage
Stop

Fig. 2. Flowchart for generating test cases.

diagram of Fig. 1. SCIG is a directed graph which is shown in Fig. 4. The purpose of the directed graph is to show the details of the traversed path.

4.1.1. All transitions (AT)

This section presents the methodology to generate test paths utilising distinctive algorithms for different coverage criteria using the case study Stack Operation. Before the generation of the test cases, all possible test paths have been generated from SCIG. AT coverage ensures all events, all states, and actions of the state chart model. The algorithm has been proposed as *All Transitions State Algorithm (ATSA)* to generate test cases (Swain and Mohapatra, 2010). Using *ATSA* algorithm, all necessary paths are extracted in the SCIG of Stack Operation. In *ATSA* steps 2–21 return the necessary paths and the corresponding test cases. In step 21, T represents the test suite. The algorithm, *ATSA*, for all transitions that generate the paths, is shown below.



```
package stack1;
public class stk
    {
    private int maxSize;
    private long[] stackArray;
    private static int top;
    public void create(int s)
    {
        maxSize=3;
        stackArray=new long[maxSize];
        top=-1;
    }
    public void push(long j){
        if(isFull())
        {
            System.out.println("Stack is Full");
            return;
        }
        stackArray[++top]=j;
    }
    public long pop(){

        if(isEmpty())
```

```
empty state
holding state
empty state
full state
holding state
holding state
20
stack destroyed
```

Fig. 3. Program implementation for Stack Operation.

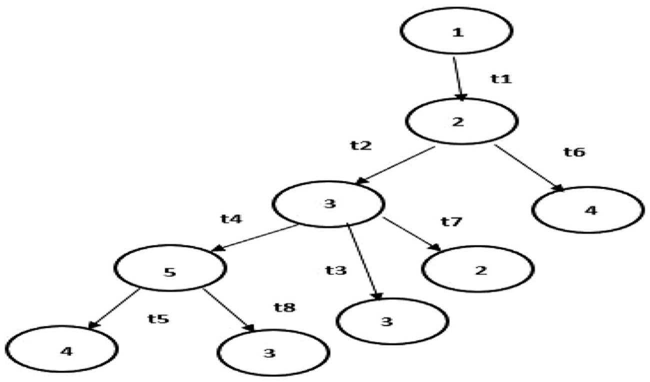


Fig. 4. State Chart Intermediate Graph (SCIG) of Stack Operation.

Algo 1: ATSA

```
Algorithm ATSA
    Input: SCIG of the state chart diagram
    Output: Sequence of transitions
Step 1: PATH [ ] = Identify all basic paths of SCIG = (PATH[1],
        PATH [2] ....PATH[n]) all paths from start to end node
        of SCIG
Step 2: For each PATH[i] $\in$ PATH do
Step 3: Current-node (C--node) = S, S is a starting node taken
Step 4: Pre $C_i$ = Find Pre Cond (S)
Step 5: $t_{i \leftarrow \varnothing}$
Step 6: while C-node $\neq$ final node of path P[i] do
Step 7: event (C-node) = Find Event(C-node)
Step 8: If C $\neq$ G // For no condition G
Step 9: t = {pre C}, I($a_1$, $a_2$, $a_3$, . . . . . ., $a_i$), O($d_1$, $d_2$, . . ., $d_m$), post C}
    //pre C = Precondition of the method
    //I ($a_1$, $a_2$, $a_3$. . . $a_i$) = set of input values for the method
    //O ($d_1$, $d_2$. . $d_m$) = set of outputs after execution of the
    method
```

Algo 1 (*continued*)

```
Algorithm ATSA
    //post C = post condition of the method
Step 10: End if
Step 11: If (C= = G) then // having guard condition for the
         method
Step 12: C (Val) = ($C_1$, $C_2$. . . $C_i$) // Set of values on the P [i] path
Step 13: t = {pre C, I ($a_1$, $a_2$, $a_3$. . . $a_i$), O ($d_1$, $d_2$. . . $d_m$), C (Val), post
         C}
Step 14: End-if
Step 15: $t = t_i \cup t$ // t is added to the set $t_i$(set of test cases)
Step 16: C-node = next-node
Step 17: End {While Final result output $O_i$ and post $C_i$ is
         determined for the end node in PATH [i]}
Step 18: t = {pre $C_i$, I, O, post $C_i$}
Step 19: $t_i \leftarrow t_i \cup t, T \leftarrow T \cup t_i$
Step 20: End-for
Step 21: Return $T$ // test suite
Step 22: Stop Process
```

Using *ATSA* algorithms, the sequence of test cases are generated based on AT coverage criteria is given below.

AT: {Test Case (TC1) = (t1, t2, t3), TC2 = (t1, t2, t7), TC3 = (t1, t2, t4, t8), TC4 = (t1, t2, t4, t5), and TC5 = (t1, t6)}.

4.1.2. Round Trip Path (RTP)

In RTP, the start and end node of each path is a single and same state. A test tree is constructed with nodes and corresponding edges to cover all paths. The nodes and the edges are created using depth-first-traverse (Briand et al., 2004; Antoniol et al., 2002). To generate the sequence of transitions for RTP coverage criteria, the algorithm has been proposed as *Round Trip Path State Algorithm (RTPSA)*. The algorithm *RTPSA* is given as Algo 2 mentioned in the dotted box.

Algo 2: RTPSA

```
Algorithm: RTPSA
Input: SCIG of state chart diagram
Output: Sequence of transitions
Step 1: Set of nodes S = [n1, n2, n3...nm] {where n1... nm are
        nodes of the state transition system}.
Step 2: Let the initial node is n1, which is the starting node and
        is the starting state of the machine.
Step 3: Traversing from $n_i$ node to all other nodes.
Step 4: Sequence till repeating the same node as $n_i$.
Step 5: Then stop the process as the node is already traversed.
Step 6: All path starting and ending with the same node $n_i$.
Step 7: Now $n_i$ is the starting node as well as the final node
        (terminal node), for the transition.
Step 8: Repeat the process until traversing each node at least
        once.
Step 9: If desired nodes are covered, then stop the process.
```

Using *RTPSA* algorithm, the sequence of test cases is generated based on RTP coverage criteria for all round trip path and simple path are given below.

RTP: {TC7 = (t1, t2, t3, t7), TC8 = (t1, t2, t3, t4, t8, t7), TC9 = (t1, t2, t4, t5), TC10 = (t1, t2, t7, t6)}.

4.1.3. All Transition Pair (ATP)

ATP says that a test suite covers all pairs of adjacent transitions. ATP can be an adjacent transition of paths length 2, length 3, and length 4 and so on. The algorithm has been proposed as *All Transition Pair State Algorithm (ATPSA)* which is used to satisfy the coverage criterion of all transition pair. The algorithm *ATPSA* is given below as Algo 3 for ATP coverage criteria (Swain et al., 2012b).

Algo 3: All Transition Pair State Algorithm (ATPSA

```
Algorithm name: ATPSA
Input: SCIG of state chart diagram
Output: Sequence of transitions
Step 1: For the all transition pair Set of nodes S = [n1, n2,
        n3...nm] {where n1... nm are nodes of the state
        transition system}.
Step 2: Let ni is the initial node.
Step 3: Move to the initial node $n_i$to $n_j$in sequence as a pair of
        adjacent transitions
Step 4: Moreover, move from node $n_j$ to $n_k$ as the adjacent
        transitions of paths of length 2 in sequence.
Step 5: Then traverse from the starting node $n_i$ to the adjacent
        nodes of adjacent paths of length 3 in sequence.
Step 6: Traverse from the starting node n1 to the adjacent
        nodes of adjacent paths of length 4 in sequence.
Step 7: If all the nodes are covered at least once with paths
        LN2, LN3, LN4 from the starting node to the final
        nodes, then
Step 7: Stop the process.
```

Utilising A*TPSA* algorithm, the sequence of test cases is generated based on ATP coverage criteria is given below.

ATP with LN2: {TC11 = (t1, t2, t3), TC12 = (t1, t2, t4), TC13 = (t1, t2, t7)}.

ATP with LN3: {TC14 = (t1, t2, t3, t7), TC15 = (t1, t2, t7, t6), TC16 = (t1, t2, t4, t5), TC17 = (t1, t2, t4, t8)}. Similarly, ATP with LN4: {TC18 = (t1, t2, t3, t4, t5), TC19 = (t1, t2, t3, t7, t6), TC20 = (t1, t2, t3, t4, t8)}.

Algo 4: Test case generation for SDVMAS

```
Algorithm for test path and test case generation based on
transitions of vending machine
Input: SCIG of the state chart diagram, RD1 (Required Drink of
       Category 1), RD2 (Required Drink of Category 2), RD3
       (Required Drink of Category 3), DP1, DP2, DP3 (Drink
       Prices for each Drink of Category 1, Category 2 and
       Category 3), Amount (Money entered)
Output: STC (Sequence of Test Cases), ATC (All Transition
        Coverage), AC (Action Coverage), All Transition Pair
        Coverage (ATPC), Round Trip Path Coverage (RTPC),
        SC (State Coverage)
Initial State: Starting State of the state machine
Current State: Present State of the state machine
End State: Final/EndState of the Transition
Begin
States of the Machine = {Idle, Showing Options To select,
  Collecting Money, Processing Order, Dispensing Drinks,
  Dispensing Change, Displaying Message, Displaying Finish
  Message}
If (machine_state is Idle)
     Output (STC, Present State, Last State)
     Showing Options To select assigns to machine_state
Step 1: If (machine_state is Processing Order)
     Output (STC, Present State, Last State)
     Showing Options To select assigns to machine_state
Step 2: If (machine_state is Showing Options To select)
     Output (STC, Present State, Final State)
     Output (AD1, AD2, AD3) [AD (Available Drinks)]
     Displaying Message assigns to machine_state
Step 3: If (machine_state is Displaying Message & Selection
        made is true)
     Output (STC, Present State, Last State)
     Output (RD1, RD2, RD3)
     Showing Options To select assigns to machine_state
Step 4: If (machine_state is Showing Options To select &
        Selection made is true)
     If ((RD1 < AD1) and (RD2 < AD2) and (RD3 < AD3))
       Output (STC, Present State, Last State)
       Processing Order assigns to machine_state
Step 5: If (machine_state is Showing Options To select)
     Output (STC, Present State, Last State)
     Collecting Money assigns to machine_state
Step 6: If (machine_state is Collecting Money)
     Output (STC, Present State, Last State)
     Processing Order assigns to machine_state
Step 7: If (machine_state is Processing Order)
     Output (STC, Present State, Last State)
     Processing Order assigns to machine_state
Step 8: If (machine_state is Processing Order)
     Output (STC, Present State, Last State)
     Processing Order assigns to machine_state
Step 9: If (machine_state is Processing Order & Selection made
        is true)
     9.1 If ((RD1 < AD1) and (RD2 < AD2) and (RD3 < AD3))
        Total money is {(RD1 × DP1) + (RD2 × DP2) +
   (RD3 × DP3)}
        Return money is Amount-Total money
        9.1.1If (Amount < Total money)
      Output ("Return money")
        Idle assigns to machine_state
        Else
```

(*continued on next page*)

Algo 4 (*continued*)

```
        Output (TS, Present State, Last State)
        Dispensing Drinks assigns to machine_state
  Step 10: If (machine_state is Dispensing Drinks and Selection
          made is true)
        Output (STC, Present State, Last State)
        Output ("Dispense Soft Drink")
        Dispensing Change assigns to machine_state
  Step 11: If (machine_state is Dispensing change and Selection
          made is true)
        Output (STC, Present State, Last State)
        Output ("Dispense Return Money")
        Displaying Finish Message assigns to machine_state
  Step 12: If (machine_state is Processing Order and Selection
          made is false)
        Output (STC, Present State, Last State)
        Displaying Finish Message assigns to machine_state
  Step 13: If (machine_state is Displaying Finish Message)
        Output (STC, Present State, Last State)
        Idle assigns to machine_state
     End
```

Sometimes the test trees generated from state machines are not deterministic using AT, RTP, and ATP coverage criteria. However, a couple of test trees could satisfy the criteria. When traversing the state machine, the tree structure relies upon the sequence of the transitions. Thus various possible transition trees are generated to detect incorrect outputs or system states.

Fig. 5 shows two transition trees (a, b) using various coverage criteria through BFS traversal. In Fig. 5, despite the two trees vary symmetrically, the first tree (a) is semantically equivalent to the second tree (b). The results of different test suites executions fulfill the same test criterion may vary because of conceivable contrasts in fault-detection power.

4.2. Case study 2 (Soft Drink Vending Machine Automation System)

This section elaborates the working process of the coverage criteria using a Soft Drink Vending Machine Automation System (SDVMAS). The state chart diagram of SDVMAS object with various events is shown in Fig. 6. When the power button is switched on, the object first enters into the *Idle* state. Three categories of soft drink type in the menu for the vending machine are taken here. Once selected drink type event is triggered with no. of soft drinks $\leq$10 (cannot deliver more than 10 soft drinks), it enters into the state *Displaying Message* and displays with the price list for different types of soft drinks. After the type and number of soft drinks are chosen, the object enters into the *Showing Options To select* state. The calculated amount of money as per the order is displayed in the *Showing Options To select* state. We have the notation as:

[Total_money = (X1*P1 + X2*P2 + X3*P3) where X1 is the no. of soft drink of Category 1, X2 is the no. of soft drink of Category 2, X3 is the no. of soft drink of Category 3, P1: Price of the soft drink of Category 1, P2: Price of the soft drink of Category 2, P3: Price of the soft drink of Category 3]

After entering the money by the customer, the object enters into *Processing Order* state. At *Processing Order* state, it calculates the return money (Return_money = Amount – total_money). If the condition is Return_money < zero, state changes to *Processing Order* to *Dispensing Change*. If the condition is Return_money $\geq$ zero, state changes to *Processing Order* to *Dispensing Drinks* and the machine delivers the required soft drinks. If the condition is Return_money > zero, state changes to *Dispensing Drinks* to *Dispensing Change*. The *Dispensing Change* state returns the balance money to the customer. After returning the balance money, the object goes to the *Idle* state. SCIG for the corresponding Vending machine automation system is shown in Fig. 7. In this case study, the number of transitions and states are 12 and 8 respectively. The algorithm for SDVMAS is given as Algo 4 in the dotted box as test case generation for SDVMAS.

4.2.1. All Transition (AT)

Fig. 7 shows the intermediate graph of the case study SDVMAS to outline the objective of all transition (AT) coverage. The sequences of the testing paths are given below.

AT: {TC1 = (t1, t2), TC2 = (t1, t3), TC3 = (t1, t3, t4), TC4 = (t1, t3, t4, t5), TC5 = (t1, t2, t6), TC6 = (t1, t5, t7), TC7 = (t1, t5, t8), TC8 = (t1, t5, t9), TC9 = (t1, t5, t7, t8, t10), TC10 = (t1, t5, t7, t8, t10, t11), TC11 = (t1, t5, t9, t11, t12)}.

4.2.2. Round Trip path (RTP)

To cover every such path in RTP, a transition tree is constructed, consisting of nodes and edges. The tree is constructed utilising breadth-or depth-first algorithm.

RTP: **{**TC12 = (t1, t3, t4), TC13 = (t1, t5, t8, t10, t11, t12), TC14 = (t1, t5, t9, t11, t12), TC15 = (t1, t2, t6, t8, t10, t11, t12), TC16 = (t1, t5, t7, t9, t11, t12)}.

Using breadth-first search traversal, the conceivable trees produce in Figs. 8–10 for the vending machine (Fig. 6) depend on following round trip path criterion.

In RTP though TREE 1 symmetrically varies to TREE 3, TREE 1 is semantically equal to TREE 3. It implicates that TREE 2 also semantically equal to both TREE1 and TREE 3. In this manner, because of conceivable contrasts in fault-detection power, the executing results of different test suites with the same criteria may be different.

4.2.3. All transition Pair (ATP)

In ATP coverage criteria, a test suite that contains tests covering all pairs of adjacent transitions from state S_i to S_j and from state S_j to S_k in the state machine.

ATP: {TC17 = (t1, t3, t4, t3), TC18 = (t1, t2, t6, t9, t11, t12), TC19 = (t1, t5, t7, t8, t10, t11)}.

ATP with LN2: {TC20 = (t1, t3, t4), TC21 = (t1, t2, t6), TC22 = (t1, t5, t7), TC23 = (t1, t5, t9), TC24 = (t1, t5, t8)}.

ATP with LN3: {TC25 = (t1, t3, t4, t2), TC26 = (t1, t3, t4, t5), TC27 = (t1, t5, t7, t8), TC28 = (t1, t5, t7, t9), T29 = (t1, t2, t6, t8), TC30 = (t1, t5, t8, t10)}.

ATP with LN4: {TC31 = (t1, t3, t4, t2, t6), TC32 = (t1, t2, t6, t7, t8), TC33 = (t1, t5, t9, t11, t12), TC34 = (t1, t5, t8, t10, t11)}.

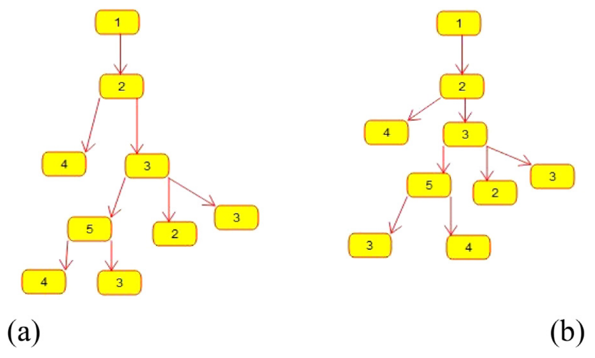


Fig. 5. Different transition trees for Stack Operation.

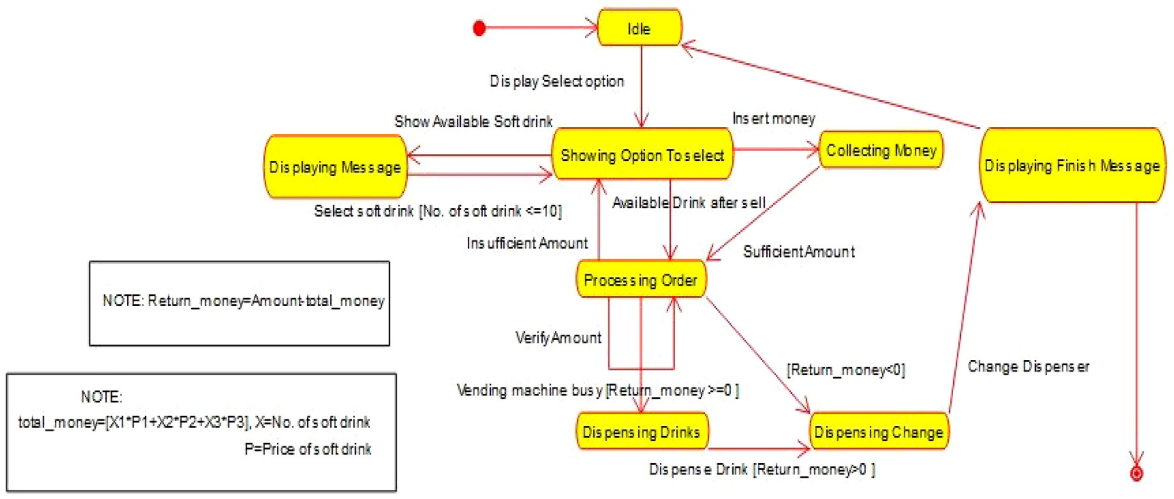

Idle
Display Select option
Show Available Soft drink
Insert money
Displaying Message
Showing Option To select
Collecting Money
Displaying Finish Message
Select soft drink [No. of soft drink <=10]
Available Drink after sell
Insufficient Amount
Sufficient Amount
Processing Order
NOTE: Return_money=Amount-total_money
Verify Amount
[Return_money<0]
Change Dispenser
Vending machine busy [Return_money>=0]
NOTE:
total_money=[X1*P1+X2*P2+X3*P3], X=No. of soft drink
P=Price of soft drink
Dispensing Drinks
Dispensing Change
Dispense Drink [Return_money>0]

Fig. 6. State chart diagram of the soft drink vending machine automation system.

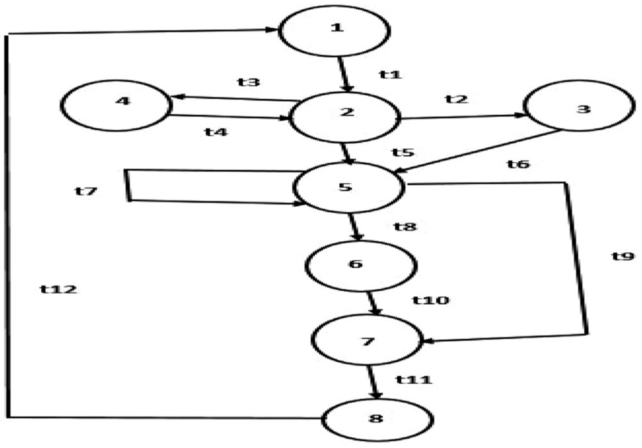


Fig. 7. Showing the corresponding SCIG of SDVMAS.

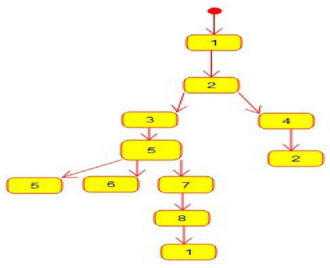


Fig. 8. TREE 1.

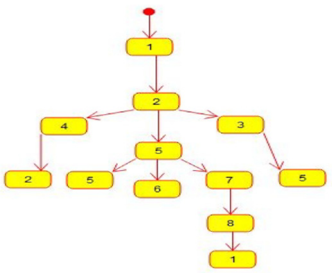


Fig. 9. TREE 2.

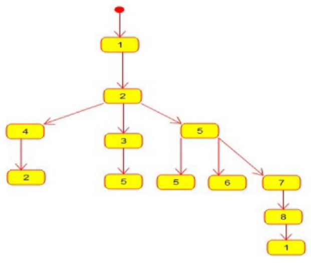


Fig. 10. TREE 3.

5. Evaluation results

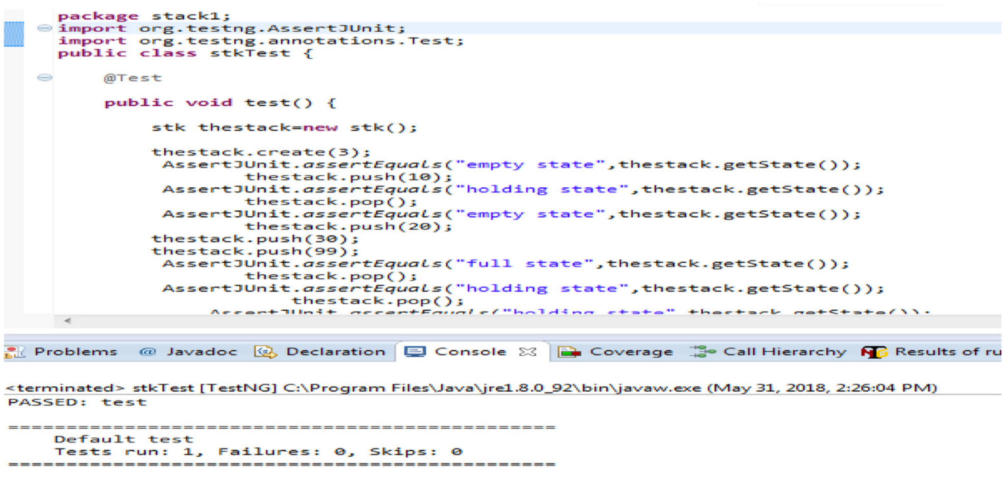
Case Study1: (Stack Operation)

The depiction of Fig. 11 demonstrates that all generated test cases pass in the TestNG structure of J-Unit. Considering the events with conditions, we find the test data for the Stack Operation which is as follows:

Test Case (TC1) = (push 10, push 20, stack holding) [stack size < max size]
TC2 = (push10, push20, push30, stack full) [stack size = max size]
TC3 = (pop 30, stack holding)[stack size < max size]
TC4 = (pop 30, pop 20, stack holding)
TC5 = (pop 30, pop 20, pop 10, stack empty)
TC6 = (stack full, stack final)

The coverage analysis of both the case studies appears in Tables 1 and 2**.** The objective is to show the maximum coverage of state and transition on SBT.

The percentage of state and transition coverage of various coverage criteria are shown in Table 1 for Stack Operation. We observe that RTP is more effective than the other coverage criteria like ATP and AT. The number of test cases is more in AT, which makes the preparation time and execution time more than that of the RTP. Even though state coverage in case of AT, RTP, ATP with LN3 and ATP with LN4 are relatively same, the significant differences are the number of test cases and the fault revealing power of the RTP with possible transition tress. The state and transition coverage of ATP with LN2 isn't sufficient for this situation. The transition coverage of ATP with LN4 might have 100% coverage.



```
package stack1;
import org.testng.AssertJUnit;
import org.testng.annotations.Test;
public class stkTest {

    @Test

    public void test() {

        stk thestack=new stk();

        thestack.create(3);
         AssertJUnit.assertEquals("empty state",thestack.getState());
                thestack.push(10);
         AssertJUnit.assertEquals("holding state",thestack.getState());
                thestack.pop();
         AssertJUnit.assertEquals("empty state",thestack.getState());
                thestack.push(20);
        thestack.push(30);
        thestack.push(99);
         AssertJUnit.assertEquals("full state",thestack.getState());
                thestack.pop();
         AssertJUnit.assertEquals("holding state",thestack.getState());
                    thestack.pop();
           AssertJUnit.assertEquals("holding state",thestack.getState());
```

```
<terminated> stkTest [TestNG] C:\Program Files\Java\jre1.8.0_92\bin\javaw.exe (May 31, 2018, 2:26:04 PM)
PASSED: test

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================
```

Fig. 11. Snapshot showing passed test cases for stack operation.

Table 1
State and transition coverage concerning various coverage criteria in Stack Operation.

Coverage Criteria	No. of Test cases	State Coverage in %	Transition Coverage in%
AT	5	100	100
RTP	4	100	100
ATP with LN2	3	80	62.5
ATP with LN3	4	100	50
ATP with LN4	3	100	100

Table 2
State and transition coverage concerning various coverage criteria in soft drink vending machine automation system.

Coverage Criteria	No. of test cases	State Coverage in %	Transition coverage in%
AT	11	100	100
RTP	5	100	100
ATP with LN 2	5	82	75
ATP with LN 3	4	100	66.6
ATP with LN 4	4	100	100

In Table 2, the state transition coverage ratios of various coverage criteria appear for the case study, SDVMAS. From the Table, we observe that RTP is more effective than the other coverage criteria like ATP and AT. The number of test cases (11) is more in AT, which makes the preparation time and execution time more than that of the RTP. RTP has 5 numbers of test cases. It is almost half of the AT. The transition coverage of ATP with LN4 might have 100% coverage.

The coverage analysis of Stack Operation (case study 1) is shown in the Bar graph of Fig. 12. The Bar graph is set utilising the information given in Table 1. The Bar graph shows 100% coverage for the state and transition in case of RTP with less number of test cases. ATP with LN4 might have 100% coverage. For the number of test cases, the preparation and execution time is more in case of ATP with LN4.

The coverage analysis of SDVMAS is shown in Bar graph of Fig. 13. The Bar graph shows 100% coverage for the state and transition for RTP with less number of test cases than AT. ATP with LN4 might have 100% coverage.

Case Study 2: (SDVMAS)

The format for SDVMAS test data is represented as {IS (Input State), TD (Test Data), EOS (Expected Output State)}. Let us have the initial data as [(5, 5, 5), 300] for X1 = X2 = X3 = 5, where the order of each drink is ≤5, and each drink price is 20 rupees. Then the Total_money = (5 × 20 + 5 × 20 + 5 × 20) = 300 and Return_money = 500–300 = 200. For the condition (Return_money > 0), the test case TC35 = {*Processing Order, (5, 500), Dispensing Drinks*, Return_money}. Considering the condition (Return_money = 0), the test case TC36 = {*Processing Order, (5, 300), Dispensing Drinks*}. Considering the boundary condition (Return_money < 0), the test case TC37 = {*Processing Order, (5, 200), Dispensing Change*, Return_money}. By triggering the events Return_money > 0, Return_money = 0 and Return_money < 0 for the test data

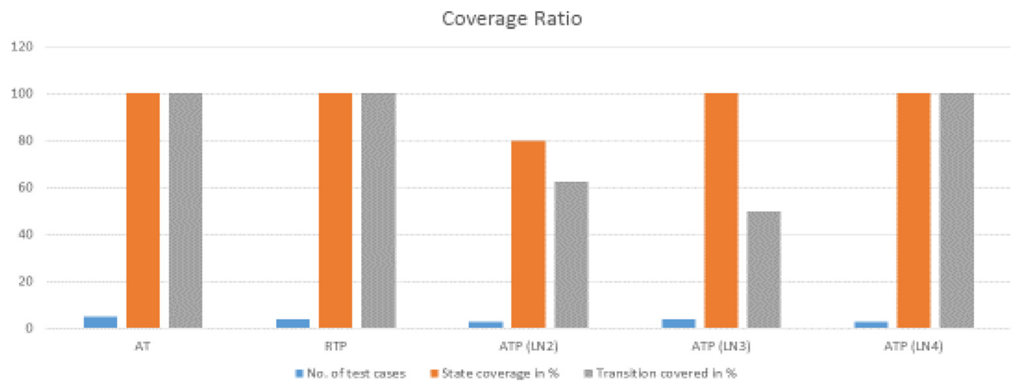


Fig. 12. Coverage analysis of Stack Operation.

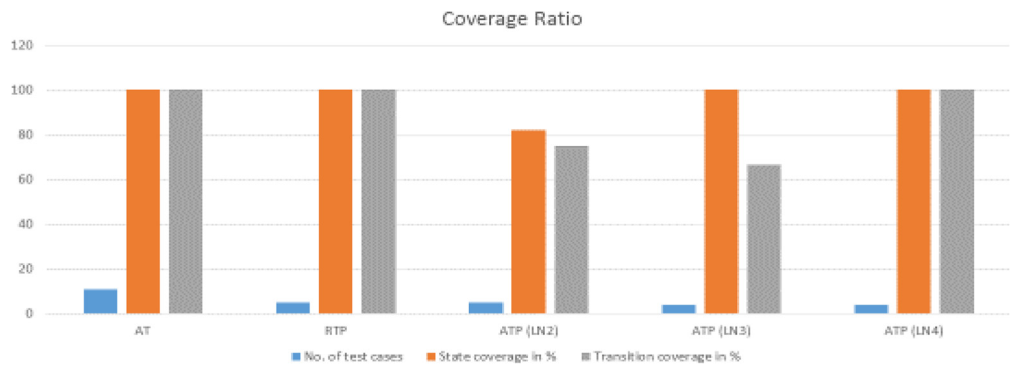


Fig. 13. Coverage analysis of soft drink vending machine.

generation, we find three test cases as TC35, TC36, TC37 and the test data are [(5, 500), (5, 300), (5, 200)] respectively.

6. Comparison study with existing work

Swain et al. (2012b) propose an algorithm to derive test cases from state chart diagram using state coverage, transition pair coverage and action coverage. The test cases are generated to minimize cost as well as time. Unlike our approach, they concentrate on only one coverage criteria, all state-transition coverage, which requires more testing resources. We propose algorithms based on various coverage criteria. Out of those, RTP can detect various state-based faults with less number of test cases. They give more emphasis on state coverage rather than transitions and also don't pay attention to redundancy. We emphasize on avoiding redundant test cases with the generation of bound and simple test paths which is not considered in their approach. We are able to overcome the transition explosion problem using SCIG. Unlike their approach, we consider K longer paths instead of all-path which is time-consuming.

Briand et al. (2004) prove that ATP with LN2 is weaker than AT and RTP. In our case, we also experimentally observe that RTP is better than ATP with LN2. RTP covers more state-based faults. We make a better understanding of RTP coverage criteria by traversing the paths in a tree. It is not always utilizing a complete round trip path. We investigate different symmetrical trees to observe possible faults. Though the trees are semantically equivalent, the test results may differ with executing the test suites.

7. Conclusion and future work

We focus on generating test cases from the state chart diagram by implementing different coverage criteria to detect various state-based faults. We analysed various state-based coverage criteria and observed that RTP is the best coverage criteria among them when the state change of the state machine is non-deterministic. Fault prediction is more in the case of coverage criteria, RTP, as compared to other coverage criteria. It is experimentally observed that the coverage criteria, AT, is most likely and not adequately reliable for an indicator of fault detection. The test suite preparation and execution time for ATP is more compared to other coverage criteria. The limitation is that we have not considered the event dependency graph. We only considered the state chart diagram. The event dependency graph is beneficial for a critical system and it is used to identify critical components, interfaces and subsystems. In future, we will include the cost-effectiveness analysis for test case generation of a critical system in SBT using the event dependency graph. The work can be extended to detect the faults related to unspecified behaviour (sneak-paths) of the state machine and how other UML diagrams can be combined with state chart diagram to achieve more number of state-based faults.

Declaration of Competing Interest

The authors declare that there is no conflict of interest regarding the publication of this article.

References

Abdurazik, A., Offutt, J., 1999. Generating Test Cases from UML Specifications Master's thesis. George Mason University.
Alhir, S.S., 2002. Introduction to the unified modeling language (UML). In: Guide to Applying the UML. Springer, New York, pp. 1–11.
Aljawarneh, S.A., Alawneh, A., Jaradat, R., 2017. Cloud security engineering: early stages of SDLC. Future Gener. Comp. Syst. 74, 385–392.
Antoniol, G., Briand, L.C., Di Penta, M., Labiche, Y., 2002. A case study using the round-trip strategy for state-based class testing. In: Proceedings of 13th International Symposium on Software Reliability Engineering. IEEE, pp. 269–279.
Bauer, B., Odell, J., 2005. UML. UML 2.0 and agents: how to build agent-based systems with the new UML standard. Eng. Appl. Artif. Intell. 18 (2), 141–157.
Bohme, M., Pham, V.T., Roychoudhury, A., 2017. Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Software Eng.
Briand, L.C., Labiche, Y., Wang, Y., 2004. Using simulation to empirically investigate test coverage criteria based on state chart. In: 26th International Conference on Software Engineering, 2004. ICSE 2004. IEEE, pp. 86–95.
Burgueno, L., Vallecillo, A., Gogolla, M., 2018. Teaching UML and OCL models and their validation to software engineering students: an experience report. Comput. Sci. Educ., 1–19
Da Silva, A.R., 2015. Model-driven engineering: a survey supported by the unified conceptual model. Comp. Languages, Syst. Struct. 43, 139–155.
El-Fakih, K., Simao, A., Jadoon, N., Maldonado, J.C., 2017. An assessment of extended finite state machine test selection criteria. J. Syst. Softw. 123, 106–118.
Fitzgerald, B., Stol, K.J., 2017. Continuous software engineering: a roadmap and agenda. J. Syst. Softw. 123, 176–189.
Hashim, N.L., Ibrahim, H.R., Rejab, M.M., Romli, R., Mohd, H., 2018. An empirical evaluation of behavioral UML diagrams based on the comprehension of test case generation. Adv. Sci. Lett. 24 (10), 7257–7262.
Holt, N.E., Briand, L.C., Torkar, R., 2014. Empirical evaluations on the cost-effectiveness of state-based testing: an industrial case study. Inf. Software Technol. 56 (8), 890–910.
Kabir, S., 2017. An overview of fault tree analysis and its application in model-based dependability analysis. Expert Syst. Appl. 77, 114–135.
Kansomkeat, S., Rivepiboon, W., 2003. Automated-generating test case using UML state chart diagrams. In: Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology South African Institute for Computer Scientists and Information Technologists, pp. 296–300.
Lonetti, F., Marchetti, E., 2018. Emerging software testing technologies. Advances in Computers, vol. 108. Elsevier, pp. 91–143.
Mathur, A.P., 2013. Foundations of Software Testing, 2/e. Pearson Education India.
Mall, R., 2018. Fundamentals of Software Engineering. PHI Learning Pvt. Ltd..
Matalonga, S., Rodrigues, F., Travassos, G.H., 2017. Characterizing testing methods for context-aware software systems: results from a quasi-systematic literature review. J. Syst. Softw. 131, 1–21.
Misra, R., Panigrahi, C.R., Panda, B., Pati, B., 2018. Software design. In: Application Development and Design: Concepts, Methodologies, Tools, and Applications. IGI Global, pp. 18–56.
Mostowski, W., 2019. Model-based fault injection for testing gray-box systems. J. Logical Algebraic Methods Programm. 103, 31–45.

Neto, A.D., Subramanyan, R., Vieira, M., Travassos, G.H., Shull, F., 2008. Improving evidence about software technologies: a look at model-based testing. IEEE Softw. 25 (3), 10–13.

Nidhra, S., Dondeti, J., 2012. Black box and white box testing techniques-a literature review. Int. J. Embedded Syst. Appl. (IJESA) 2 (2), 29–50.

Offutt, J., Liu, S., Abdurazik, A., Ammann, P., 2003. Generating test data from state-based specifications. Software Test., Verif. Reliab. 13 (1), 25–53.

Pradhan, S., Ray, M., Patnaik, S., 2019. Coverage criteria for state-based testing: a systematic review. Int. J. Inf. Technol. Project Manage. (IJITPM) 10 (1), 1–20.

Prasanna, M., Chandran, K.R., 2009. Automatic test case generation for UML object diagrams using genetic algorithm. Int. J. Adv. Soft Comput. Appl 1 (1), 19–32.

Pressman, R.S., 2005. Software Engineering: A Practitioner's Approach. Palgrave Macmillan.

Ray, M., Mohapatra, D.P., 2012. Prioritizing Program Elements: A Pretesting Effort to Improve Software Quality. ISRN Software Engineering.

Saeed, A., Ab Hamid, S.H., Mustafa, M.B., 2016. The experimental applications of search-based techniques for model-based testing: taxonomy and systematic literature review. Appl. Soft Comput. 49, 1094–1117.

Salman, Y.D., Hashim, N.L., 2016. Automatic test case generation from UML state chart diagram: a survey. In: Advanced Computer and Communication Engineering Technology. Springer, Cham., pp. 123–134.

Sharma, A., Misra, P.K., 2017. Aspects of enhancing security in software development life cycle. Adv. Comp. Sci. Technol. 10 (2), 203–210.

Shafique, M., Labiche, Y., 2010. A systematic review of model based testing tool support. Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04, 01-21.

Swain, R., Panthi, V., Behera, P.K., Mahapatra, D.P., 2012a. Test case generation based on state machine diagram. Int. J. Comp. Inf. Syst. 4 (2), 99–124.

Swain, R. K., Behera, P. K., & Mohapatra, D. P., 2012b. Minimal test case generation for object-oriented software with state charts. arXiv preprint arXiv:1208.2265.

Swain, R., Panthi, V., Behera, P.K., Mohapatra, D.P., 2012c. Automatic test case generation from UML state chart diagram. Int. J. Comp. Appl. 42 (7), 26–36.

Swain, S.K., Mohapatra, D.P., 2010. Test case generation from Behavioral UML Models. Int. J. Comp. Appl. 6 (8), 5–11.

Utting, M., Pretschner, A., Legeard, B., 2006. A taxonomy of model-based testing.

Utting, M., Pretschner, A., Legeard, B., 2012. A taxonomy of model-based testing approaches. Software Test., Verif. Reliab. 22 (5), 297–312.

Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A., 2016. Recent advances in model-based testing. Advances in Computers 101, 53–120.

Zhou, Z.Q., Sinaga, A., Susilo, W., Zhao, L., Cai, K.Y., 2018. A cost-effective software testing strategy employing online feedback information. Inf. Sci. 422, 318–335.