Int J Syst Assur Eng Manag (June 2017) 8(2):242–252
DOI 10.1007/s13198-016-0419-1

ORIGINAL ARTICLE

Generating and evaluating effectiveness of test sequences using state machine

Vikas Panthi[1] · Durga Prasad Mohapatra[1]

Received: 18 August 2015 / Revised: 19 January 2016 / Published online: 3 February 2016

Abstract The aim of this paper is to generate test sequences for object-oriented software with composite states using state machines. This experimental work in software testing focuses on generating test sequences using the proposed algorithm called SMTSG (State Machine to Test Sequence Generation). This work also describes the effectiveness of test sequences by using mutation analysis. Our approach considers nine types of state faults for checking the efficiency of the generated test sequences. The effectiveness of the prioritized test sequences is shown using Average Percentage Fault Detection (APFD) metric. The experimental results show that the test sequences generated using our proposed approach are more efficient than the existing approaches.

Keywords Test sequences · Average Percentage Fault Detection (APFD) Metric · State Machine Diagram (SMD)

1 Introduction

In a typical software project, more than 60 % of the effort is spent on software testing to produce reliable software. So, adequate testing is one of the most demanding activity in software development process. This is the reason why it is receiving increased attention for researchers. Test case generation techniques, based on source code, are very cumbersome. This is especially true for complex and large projects. Further, test cases generated from source code, are inadequate in case of component based software. Exhaustive testing of software is very time-consuming and error-prone. The design based test cases are used to overcome the above mentioned limitations. Hence, model-based test sequence generation is one of the emerging trends in object-oriented software testing (Chen et al. 2008; Jorgensen 2008).

UML state machine diagrams are often used to describe the behavior of a system. State machine diagram contributes a significant role in system and integration testing. Therefore, automatic test sequence generation using state machine diagram is implemented for testing the system requirements.

The problems: UML-based automatic test sequence generation is both practically and theoretically challenging. Test sequence generation and execution are time-consuming and labor-intensive activities. Hence, automatic test sequence generation is an imperative need among the research community (Punuganti et al. 2007; Shirole and Kumar 2013). Test sequences must be executed systematically to detect highest possible number of faults (Bernhard et al. 2014; Elbaum et al. 2002; Gupta and Jalote 2008; Pandey and Shrivastava 2011).

The presence of object-oriented features such as polymorphism, inheritance, dynamic binding, etc. enhance the benefits of modeling, extensibility and usability (Jorgensen 2008). These features help in improving the quality of software. On the other hand, these features introduce new types of faults that do not exist in traditional procedural software (Kim et al. 2001). Several faults in object-oriented programs such as *behavioral faults, interaction faults,*

✉ Vikas Panthi
512CS103@nitrkl.ac.in; vpanthi@gmail.com

Durga Prasad Mohapatra
durga@nitrkl.ac.in

1 Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India

polymorphic faults are extremely hard to detect using static code analysis techniques (Bernhard et al. 2014; Pandey and Shrivastava 2011). However, these faults are easily detectable during the design phase. Generation of efficient test sequences for detecting these faults is a demanding requirement. Hence, the above problems in test sequence generation are very crucial and challenging in software development process.

The proposed solution: In this paper, we propose an algorithm called SMTSG (State Machine to Test Sequence Generation) for generating test sequences for object-oriented software. This algorithm uses the state machine diagram to generate test sequences and to verify the behavior of object-oriented software. Then, it detects the state faults which is discussed in Sect. 2. After that, we have calculated the efficiency of the generated test sequences using APFD (Average Percentage Fault Detection) Metric (Elbaum et al. 2002; Pandey and Shrivastava 2011; Punuganti et al. 2007; Rothermel et al. 2001). In the rest of the paper, we use the terms test sequences and test scenarios interchangeably.

The rest of the paper is structured as follows. Section 2 provides an overview of UML state machine diagrams, state fault models, test case prioritization using APFD Metric. Section 3 presents our proposed approach for test scenario generation using state machine diagram. Section 4 discusses a case study named Bank ATM System. Section 5 presents an empirical method for prioritization of generated test sequences using APFD Metric. In Sect. 6 we compare our work with some existing related works. Finally, Sect. 7 concludes the paper with an insight to the future work.

2 Preliminaries

In this section, we provide an overview of basic definitions which are used in our proposed approach.

2.1 UML state machine models

State machine diagram is a combination of states, events, transitions, guards, and actions. It represents the relationship between many objects and also represents the behavior of either an object or the entire system. There are mainly two building blocks in state machine diagram: *states* and *transitions*. A state is a condition of an instance over the course of its life. It satisfies some conditions, performs some action, or waits for some event. A state may encapsulate one or many sub-states (Gerhard 2005; Shirole and Kumar 2013). In such a case, a state is called a composite state. A simple state does not have sub-states, and the composite state has some nested or hierarchy of states. A special state, namely *starting state*, shows the first condition throughout the life cycle of an instance. Another special state, namely *end state*, indicates the last condition throughout the life cycle of an instance. An object can reach its *end state* when it is destroyed. Systems without an *end state* run forever. Hence, the system is said to be in the self-transition state. A transition shows a change of state in response to an event. The event on each transition occurs due to the change of state. In state machine diagram, guard conditions are associated with the transition. There are three kinds of actions: entry, exit, and transition. A state machine model may also have some pseudo states: *initial state, fork, join, Junction* and *choice* state. The transition is shown as a directed edge from a source state to a target state. A self-transition has same state as source and target. A transition is labeled by a *sequence of actions*, *trigger event* and a *boolean guard*. The transition is defined in the form of *trigger[guard]/actions* (Shirole and Kumar 2013). In a state machine diagram, an *event* is defined as the occurrence of a stimulus that can trigger a state transition. A detailed description of a state machine diagram is available in (Shirole and Kumar 2013). A state machine can be defined as follows:

A state machine diagram is a tuple $SD = (S, S_0, E, T, S_f)$ where,

- S is a finite set of states
- $S_0 \in S$ is an initial state
- E is a finite set of events
- $T \subseteq S \times E \times S$ is a finite set of transitions
- $S_f \subseteq S$ is a finite set of final states

A state machine diagram represents the dynamic behavior of individual class objects, use cases, and entire system. For example, Fig. 2 represents a state machine diagram of a Bank ATM system.

2.2 UML state faults

Some of the possible model-based faults (Ammar et al. 2001; Bernhard et al. 2014; Punuganti et al. 2007; Usman and Peng 2008) are as follows:

- State machine based faults.
- Activity diagram based faults.
- Sequence diagram based faults.

This paper focuses only on the state machine based faults. Some of these faults are given below:

1. State diagram based faults:
   a. A missing state diagram:
   b. Interchanged diagrams:

2. State based faults:
   a. Incorrect initial state:
   b. Incorrect final state(s):
   c. Interchanged states:
   d. Missing states:
3. Transition based faults:
   a. Incorrect trigger:
   b. Interchanged transitions:
   c. Missing transitions:
4. Message based faults:
   a. Missing send messages:
   b. Corrupted attributes:
5. Variable based faults:
   a. Corrupted initial value:
   b. Corrupted dynamic value:

2.3 Test case prioritization using APFD metric

Test case prioritization schedules the test cases according to their priorities. The test cases with highest priorities are executed earlier in the regression testing process than the test cases with lower priorities (Rothermel et al. 2001; Srivastava 2008). Test case prioritization techniques arrange test cases according to execution order on the basis of some criterion. The purpose of these prioritization techniques is to increase the efficiency of test cases for regression testing method. Generally, test case prioritization problem can be defined as follows:

Test case prioritization problem: Given *T*, a test suit; *PT*, the set of permutations of *T*;. *f*, a function from *PT* to the real numbers.

Problem: *Find* $T'' \in PT$ *such that* $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Here, *PT* represents the set of all possible orderings of *T*, and *f* is a function which is applied to any such ordering.

There are many possible probabilities for prioritization using the test suite for detecting the faults during the test case execution (Rothermel et al. 2001; Srivastava 2003, 2008). Let T_s be a test suite which contains *n* test cases, and let *F* be a set of *m* faults captured by T_s. Let TF_i be the first test case in scheduling T' of *T* that detects the fault *i*. According to (Rothermel et al. 2001; Srivastava 2008), the *APFD* for test suite T' could be given by Eq. 1:

$$\mathrm{APFD} = 1 - \{(\mathrm{Tf}_1 + \mathrm{Tf}_2 + \cdots + \mathrm{Tfm})/\mathrm{mn}\} + (1/2\mathrm{n}) \tag{1}$$

APFD value ranges from 0 to 100. An ordered test suite with higher *APFD* value has faster (better) fault detection rates than those with lower *APFD* values.

Test case prioritization technique can address many important objectives including the followings:

– To increase the average percentage of faults detection, that is useful for revealing faults earlier for the execution of regression tests.
– To increase the high-risk faults detection rate in the testing process.
– To increase the probability of revealing regression errors related to mutation fault detection techniques.
– To increase the percentage of code coverage in the SUT (System Under Test) at a faster rate.
– To increase the percentage of confidence in the reliability of the SUT (System Under Test) at a faster rate.

2.4 XMI

XMI stands for XML Metadata Interchange. It is a standard representation used for exchanging metadata information by means of *EXtensible Markup Language (XML)*. The XML file is a very large file. XMill can transform XML into XMI (XML Metadata Interchange) file and store it in a compressed format. In this format, the XMI file size is around half the size of other compression techniques. XMI files can be decompressed using XMill or similar XML compression software.[1, 2]

3 Proposed approach

In this section, we discuss our proposed approach for automatically generating test sequences using state machine diagram. We have named our scheme State Machine based Test Sequence Generation (SMTSG). The schematic representation of our approach is shown in Fig. 1.

We explain all steps in detail in the following sections. We also illustrate each step with a running example of Bank ATM System given in Fig. 2.

3.1 Construct the state machine diagram for the given system and export into XMI representation

We consider an XMI representation of state machine diagram as input to our approach SMTSG. The state machine diagram is modeled using UML 2.0 metamodel specification. We construct a state machine diagram of the given

1 http://whatis.techtarget.com/.

2 http://www.w3.org/XML/.

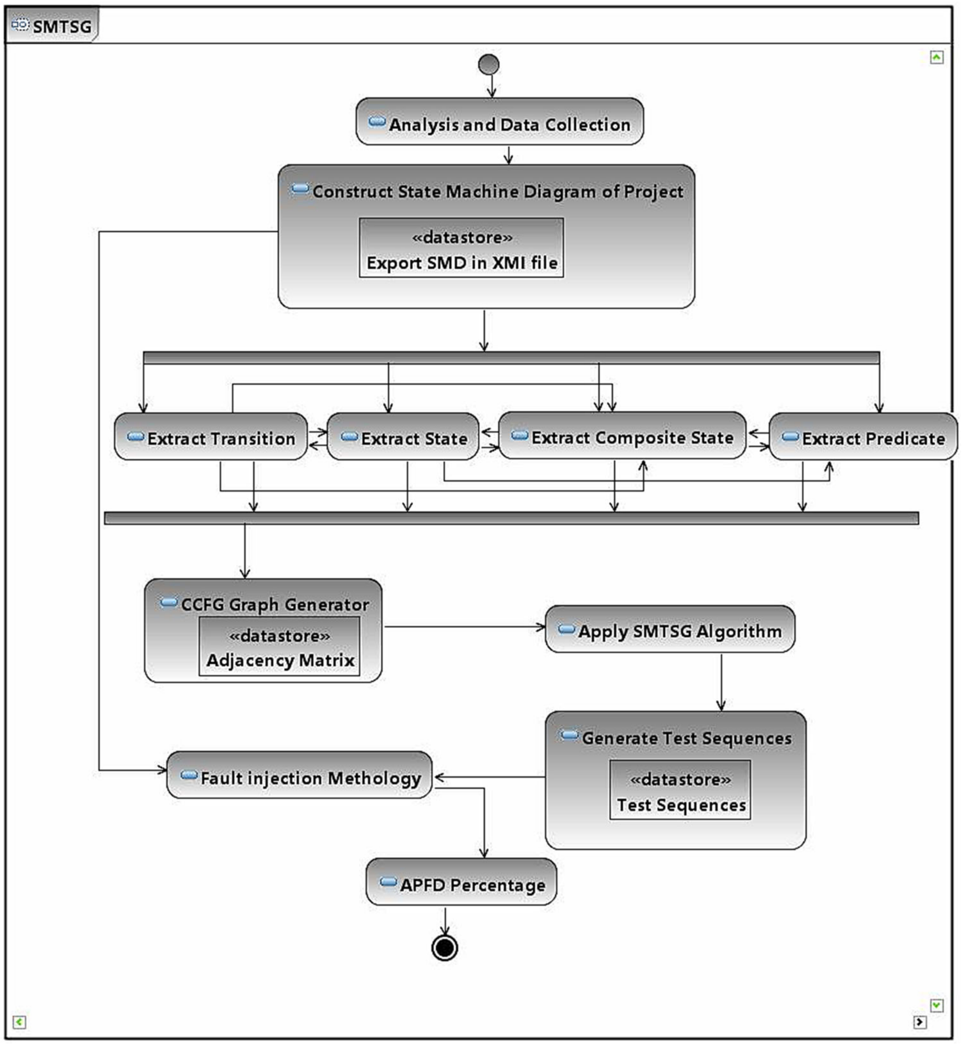


Fig. 1 Activity diagram of our proposed approach

system using IBM RSA (Rational Software Architecture) tool and export it into an XMI file.

3.2 Convert the state machine diagram to Composite Control Flow Graph (CCFG) and Adjacency matrix

We extract different element tags such as states, transitions, guard conditions, and composite states from Bank ATM system. By using the extracted element tags, we convert the state machine diagram into a CCFG (Composite Control Flow Graph). The algorithm for CCFG Generator is given in Algorithm 1. For implementing this algorithm, we develop a *XMLG (XML to Graph)* parser. This parser stores the above mentioned information in graphical and matrix form. The CCFG depicts the connectivity between the nodes. We have used CCFG as the intermediate form of the parsed state machine for our subsequent steps. Now, we explain our proposed algorithm *CCFG Graph Generator*. We supply the SMD (State Machine Diagram) in XMI format as input. The algorithm generates CCFG and adjacency matrix of state machine diagram as output. First, we create the start node S_{in}. After that, we transfer the

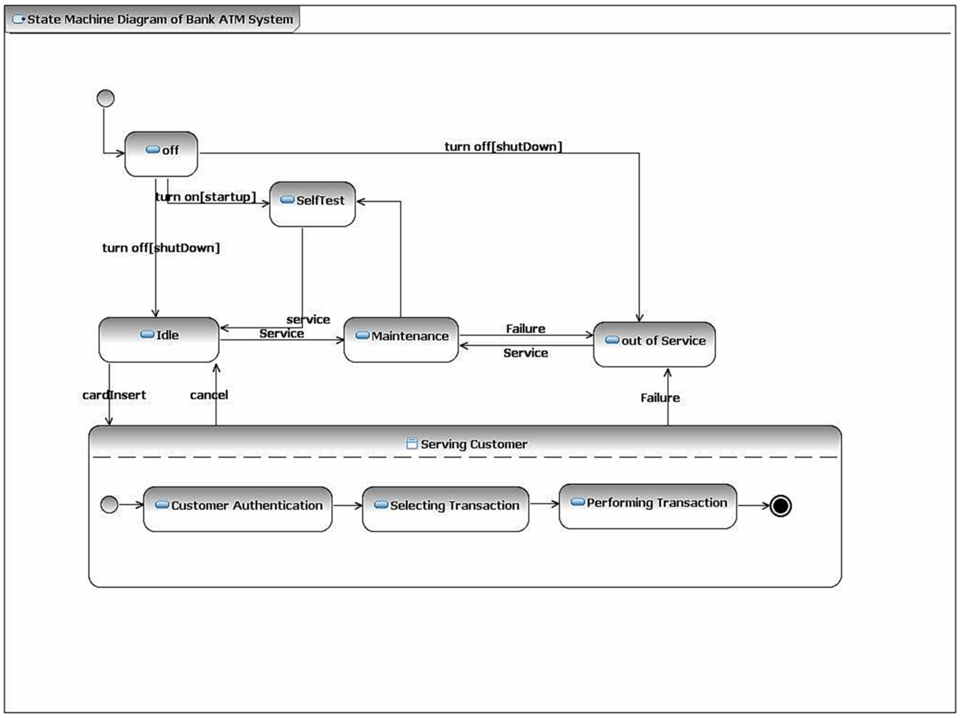


Fig. 2 State machine of Bank ATM system

current state of SMD into a temporary variable X_s. We check *if* condition $X_s = S_f$ i.e. current state X_s not equal to final state. If the condition is true, then we add the current state into *Xs*. If the condition $(X_s == C_s)$ is true, then we copy the current state into X_s. If $(X_s == S_f)$ is true, then control will be transferred to Step 2. Finally, we display the CCFG using Graphviz and store the adjacency matrix of CCFG.

Algorithm 1 *CCFG Generator*

```
Input: State Machine Diagram (SMD) in XMI format
Output: CCFG[][]
 1: Create start node $S_{in}$, CCFG[][] = φ
 2: $X_s$ = currentState(SMD, $S_{in}$); //where $S_{in}$ = Initial state and $X_s$
    = Temporary variable for current state.
 3: if $X_s \neq S_f$ then
 4:     CCFG[][]=current.State($X_s$);
 5:     $X_s = X_s \rightarrow next.state$
 6:     if $X_s == C_s$ then //where $C_s$ = Current state
 7:         CCFG[][]=current.State($X_s$);
 8:         $X_s = X_s \rightarrow next.state$
 9:         if $X_s == S_f$ then //where $S_f$ = Final state
10:             Go To Step 2.
11:         end if
12:     end if
13: end if
14: return CCFG with start node $S_{in}$ and end node $S_f$, where,
    $S_{in}, S_f \in$ SMD
15: stop
```

3.3 Generate test sequences using SMTSG algorithm and transform the test sequences into independent paths

In this step, we generate all the possible test sequences using SMTSG algorithm. The generated test sequences do not cover repeated states. But, for the execution purpose, we need all the repeated states. So, for this reason, we transform the generated test sequences into independent paths. The generated test sequences for ATM state machine are given in Table 2 and the transformed independent paths are given in Table 3.

Now, we explain our proposed algorithm SMTSG, which is given in Algorithm 2. In this algorithm, we consider the generated CCFG and Matrix as input. This algorithm generates test sequences as output. We initialize r, c, *TS*, where r = no. of rows, c = no. of columns and *TS* = Test Sequences. After that, we initialize rec[] a new array, with boolean values *false*. We trace ar[r][c] adjacency matrix and *TS* with current state of *CCFG*. Then, we check ar[r][c] row-wise and insert the current state into *TS*. Finally, we display the generated test sequences using STS (Set of Test Sequences).

This approach considers the guard conditions, composite states, and event with transitions, etc. for generating the test sequences. It may be noted that path coverage is a stronger coverage criterion than the state coverage and transition coverage criteria (Swain et al. 2010). It is ensured that all user inputs (covered solely by the transition coverage criterion) and all object responses (covered solely by the state coverage criterion) are covered by our approach. Therefore, the generated test sequences confirm the adequacy of state-transition path coverage.

Algorithm 2 *SMTSG (State Machine based Test Sequence Generation)*

```
Input: ar[r][c], r, c
Output: Set of test sequences (STS)
1: $r = 0, c = 0, TS_i[] = 0$ //where r = Rows in Adjacency Matrix of CCFG, c = Columns in Adjacency Matrix of CCFG, $TS_i$= Set of Test sequences
2: Boolean rec[]= {false, false, false..........false}
3: for i = 0 to r-1 do
4:     while $ar[r][c] = 1$ do //where $ar[r][c]$ = Adjacency Matrix of CCFG
5:         Update $TS_i = TS_i \cup CS_i$ //where $CS_i$= Current state of CCFG
6:         ar[a][b]=0
7:         rec[r]=true
8:         r=c
9:         c = -1
10:    end while
11:    while $ar[r] = true$ do
12:        Update $TS_i = TS_i \cup CS_i$//where $CS_i$= Current state of CCFG
13:        r = -1
14:        rec[r]=true
15:        c =col
16:    end while
17: end for
18: STS = $\cup_{i=1}^{N} TS_i$ //STS = Set of Test Scenarios
19: Exit
```

3.4 Prioritize the test sequences by identifying the model faults in the state machine diagram

In this step, first we execute the generated test sequences. In this process, we find some model faults which are given in Sect. 2.2. Test sequence prioritization includes scheduling the test sequences in a sequential manner. It is used to improve performance of the regression testing. After completing the process of test sequence generation, we prioritize the test sequences according to their priorities. In this proposed approach, we prioritize the test sequences according to their fault detection capability. For prioritization, we execute the test sequences and detect the faults. We find out, which test sequences detect how many faults in state machine diagram. If any test sequence detects more number of faults, then, we give it the first priority. According to this method, we prioritize all the generated test sequences. In the illustrated case study (Bank ATM System given in Fig. 2), It contains nine faults (randomly we have taken), which are detected by the generated test sequences. Prioritization of test sequences according to detected faults for the given case study (Bank ATM System given in Fig. 2) is given in Table 4. The faults are *incorrect initial state*, *incorrect final state*, *interchanged state, missing states, interchanged diagram, missing composite state, corrupted attribute, corrupted transition values.*

3.5 Find the APFD for the prioritized test sequences

We apply the generated test sequences on state machine diagram and detect some significant faults which are given in Sect. 2.2. We prioritize the test sequences according to their fault detection capability. This approach measures the effectiveness of generated prioritized test sequences. The presented approach uses a metric, called Average Percentage Faults Detection (APFD), which measures the weighted average of the percentage of faults in state machine diagram. The detailed description with illustrated case study is given in Sect. 5.

4 Case study

We consider the case study of a Bank ATM[3] system to explain our proposed approach. In Bank ATM system, there are many use cases such as check balance, withdraw cash, change PIN, transfer funds, maintenance, repair, etc. ATM system is a very large and complex system. In our case study, we consider *Maintenance* use case of ATM system. The state machine diagram of our ATM system is shown in Fig. 2. Below, we describe the state machine diagram of our Bank ATM system involving *Maintenance* use case.

Initially, ATM is in *Turned off* state. When the power is turned on, ATM performs *startup* action through *turn on* transition and enters into *Self Test* state. If *turned on* state calls *turn off* transition, then, it performs *Shut Down* action and enters into *out of service* state. According to trigger, *off* state may be entered into *Idle* state through *Shut Down* action. In *Idle* state, ATM waits for customer interaction. When the customer inserts ATM debit card in the card reader slot, the ATM state changes from *Idle* to *Serving Customer* state. *Serving customer* state is a composite state with sequential sub-states *customer authentication, selecting transaction* and *performing transaction*. *Customer authentication* state can verify authenticity of customer by the use of personal information and PIN number which are stored in the ATM debit card. *Selecting transaction* state gives the available options for customer transaction. *Performing transaction* state performs the transaction which is selected by customer. *Selecting transaction* and *Performing transaction* states depend on

[3] http://www.uml-diagrams.org/bank-atm-UML-state-machine-diagram-example.html.

Table 1 Mapping table for the state machine of Bank ATM system

SI no.	Current state		Next state		Transition	
	State	Alias state	State	Alias state	Transition	Alias transition
1.	Start	S_0	Off	S_1	–	T_0
2.	Off	S_1	Self Test	S_2	Turn on/startup	T_1
3.	Self Test	S_2	Idle	S_3	–	T_1
4.	Idle	S_3	Serving Customer	S_6	CardInserted	T_3
5.	Serving Customer	S_6	Idle	S_3	Cancel	T_4
6.	Maintenance	S_3	Maintenance	S_4	Service	T_5
7.	Out of Service	S_4	Out of Service	S_5	Failure	T_6
8.	Self Test	S_5	Maintenance	S_4	Service	T_7
9.	Serving Customer	S_2	Out of Service	S_5	Failure	T_8
10.	Serving Customer → start	S_6	Out of Service	S_5	Failure	T_9
11.	Customer Authentication	S_6	Customer Authentication	S_{61}	–	T_{10}
12.	Selecting Transaction	S_{61}	Selecting Transaction	S_{62}	–	T_{11}
13.	Performing Transaction	S_{62}	Performing Transaction	S_{63}	–	T_{12}
14.	Idle	S_{63}	Serving Customer	S_6	–	T_{13}
15.	Out of Service	S_3	Off	S_1	Turn off/shutdown	T_{14}
16.	Maintenance	S_5	Off	S_1	Turn off/shutdown	T_{15}
17.		S_4	Self Test	S_2	Turn on/startup	T_{16}

the customer interaction. *Customer authentication*, *selecting transaction* and *performing transaction* states are composite sub-states. The composite states are indicated with hidden decomposition icons. *Serving customer* state is completed when *end* state is called in composite state. If *Serving customer* state has performed action *ejectCard*, then, ATM releases customer's card on leaving the state. On entering the *serving customer* state, the *entry* action is performed, and *read card* transition is called. The *serving customer* state, moves back to the *Idle* state, when cancel transition called. The customer can *cancel* the trigger at any time.

If any problem happens on *Idle* state, then *service* transition is called and the system enters into *Maintenance* state. Finally, if any problem happens, failure transition is called and the system enters into *out of service* state.

The mapping table of alias names of different states of Bank ATM system is given in Table 1. The CCFG of the State Machine diagram given in Fig. 2 is shown in Fig. 3. The mapping table of Bank ATM System represents the alias name of states, transitions, events, etc. The mapping table is useful for generating the CCFG graph and Adjacency matrix. In this table, states start from S_0 to S_6. There are some sub-states in state machine diagram of Bank ATM system, like *customer authentication, selecting transaction* and *performing transaction*. These states are sub-states of *Serving Customer*. So, we have given alias name S_{61} to S_{63}, for S_6 (*Serving Customer*). In Table 1, we have given the current state and next state.

First, we construct the state machine diagram of Bank ATM System as shown in Fig. 2. After that, we export XMI file of state machine diagram of Bank ATM System using IBM RSA (Rational Software Architecture). XMI file is the textual representation of state machine diagram. This file stores all graphical information in text format. We store states, transitions and actions in the form of ID number. Source states and transitions states for every transition are also stored in the form of ID number. We parse the XMI file and calculate the number of nodes and edges. This information is useful for connecting different states. Using this information, we construct the CCFG (Composite Control Flow Graph). We have developed a parser named CCFG Generator. This parser generates CCFG of the state machine diagram. The generated CCFG of Bank ATM System is given in Fig. 3. The parser uses Algorithm 1 for constructing the CCFG. After generating the CCFG, we store it in an adjacency matrix. The adjacency matrix for the CCFG given in Fig. 3 is shown in Fig. 4. The adjacency matrix is used for generating the test sequences using SMTSG algorithm given Algorithm 2. The generated test sequences from the adjacency matrix of Fig. 4 are shown in Table 2. After generating the test sequences, we transform them into independent test sequences. The converted independent test sequences are given in Table 3. Besides, the Bank ATM case study, we have taken five more case studies to validate our approach. The characteristics of these case studies are given in Table 5. The generated test sequences for these case studies are shown in Table 6.

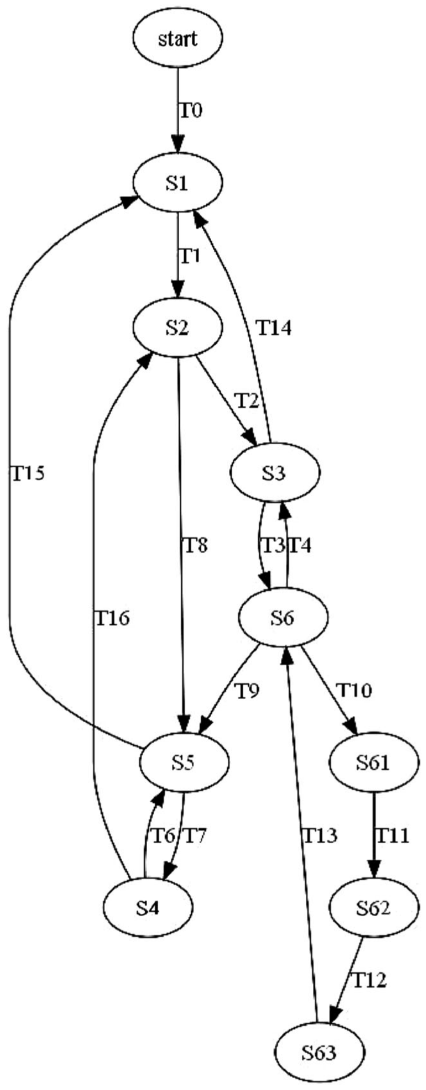


Fig. 3 Composite control flow graph of Bank ATM system

		To								
		S_1	S_2	S_3	S_4	S_5	S_6	S_{61}	S_{62}	S_{63}
From	S_1	0	1	0	0	0	0	0	0	0
	S_2	0	0	1	1	1	0	0	0	0
	S_3	1	0	0	0	0	1	0	0	0
	S_4	0	1	0	0	1	0	0	0	0
	S_5	1	0	0	1	0	0	0	0	0
	S_6	0	0	1	0	1	0	1	0	0
	S_{61}	0	0	0	0	0	0	0	1	0
	S_{62}	0	0	0	0	0	0	0	0	1
	S_{63}	0	0	0	0	0	1	0	0	0

Fig. 4 Adjacency matrix for the composite control flow graph in Fig. 3

For the Bank ATM system given in Fig. 2, suppose the system contains nine faults (randomly we have taken), which are detected by the generated test sequences is

Table 2 Test sequences for state machine of Bank ATM system generated from matrix shown in Fig. 4

TS	Optimal test sequences (OTS)
TS_1	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1$
TS_2	$S_2 \rightarrow S_5 \rightarrow S_1$
TS_3	$S_3 \rightarrow S_6 \rightarrow S_3$
TS_4	$S_4 \rightarrow S_2$
TS_5	$S_4 \rightarrow S_5 \rightarrow S_4$
TS_6	$S_6 \rightarrow S_5$
TS_7	$S_6 \rightarrow S_{61} \rightarrow S_{62} \rightarrow S_{63} \rightarrow S_6$

Table 3 Independent test sequences for Bank ATM System case study

TS	Independent test sequences
TS_1	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_1$
TS_2	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5 \rightarrow S_4$
TS_3	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_6 \rightarrow S_5$
TS_4	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_6 \rightarrow S_3$
TS_5	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_{61} \rightarrow S_{62} \rightarrow S_{63}$
TS_6	$S_0 \rightarrow S_2 \rightarrow S_4 \rightarrow S_2 \rightarrow S_5 \rightarrow S_1$
TS_7	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_6 \rightarrow S_{61} \rightarrow S_{62} \rightarrow S_{63} \rightarrow S_6$

shown in Table 3. The faults are *incorrect initial state*, *incorrect final state*, *interchanged state, missing states, interchanged diagram, missing composite state, corrupted attribute, corrupted transition values*. We have implemented our approach using Java and snapshot of implementation given in Fig. 5. Figure 5 shows the generated test sequences that detect these nine faults. Since the complete test sequences are not clearly depicted in Fig. 5. So, we have given a complete list of test sequences in Table 3.

Table 4 shows the number of faults detected by each test sequence in the test suite for Bank ATM system. There are seven test sequences, TS_1 to TS_7 in the test suite. In Table 4, we assume that there are nine faults in the state machine diagram of the Bank ATM system, which are detected by the seven test sequences. From Table 4, according to faults detection technique, it can be observed that the test sequences can be prioritized in the following order:

$$TS_7 \rightarrow TS_6 \rightarrow TS_5 \rightarrow TS_3 \rightarrow TS_4 \rightarrow TS_2 \rightarrow TS_1$$

After test sequence prioritization, we measure the effectiveness of generated prioritized test sequences. The presented approach uses a metric, called Average Percentage of Faults Detected (APFD), which measures the weighted average of the percentage of faults in state machine diagram. The detailed description of APFD with the Bank ATM system case study is given in Sect. 5.

Table 4 Prioritized test sequences with the detected faults

Test sequences/faults	Priority	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
TS_7	1	X						X		
TS_6	2			X	X					
TS_5	3		X				X			
TS_3	4									X
TS_4	5					X			X	
TS_2	6		X							
TS_1	7		X		X		X		X	X

Table 5 Characteristics of the case studies

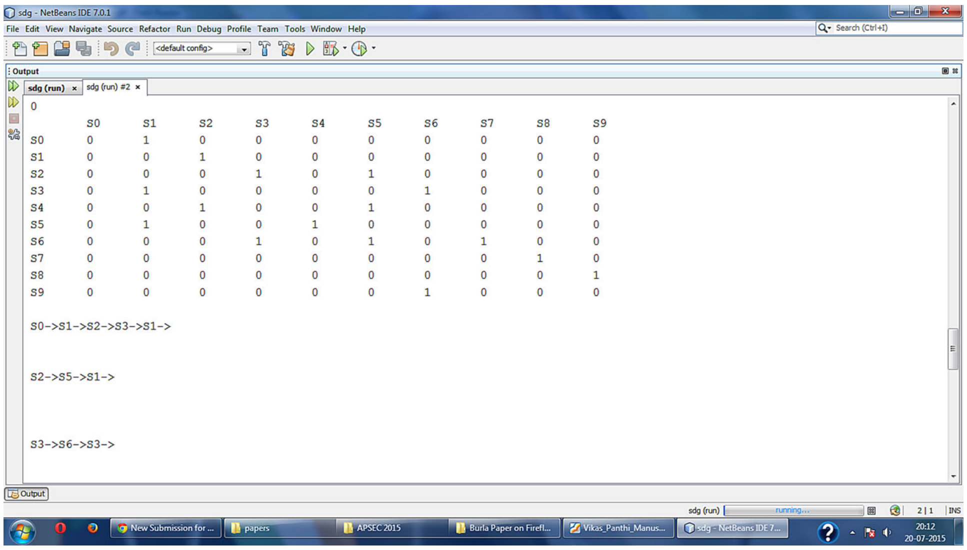
SI. no.	Case study	No. of states	No. of transitions	No. of composite states
1.	Cashier	12	21	1
2.	Cruise control	5	17	1
3.	Elevator system	6	12	1
4.	TCP	12	56	3
5.	Vending machine	7	28	1
6.	ATM	10	17	1

Table 6 Number of test sequences for the different case studies

SI. no.	Case study	No. of generated test sequences
1.	Cashier	11
2.	Cruise control	14
3.	Elevator system	8
4.	TCP	46
5.	Vending machine	23
6.	ATM	7

5 Finding APFD value of the prioritized test sequences

Scheduling the test scenarios in execution order according to some coverage criterion is called test case prioritization. The criteria may be to record test cases in an execution order that achieves maximum code coverage at the fastest rate. Test sequence prioritization is a regression testing approach. It aims at sorting and executing test cases in the order of their potential abilities to achieve certain testing objective. Rothermel et al. (2001) first introduced the prioritization problem as a flexible method of regression testing. In their technique, they selected test cases according to the modified code coverage and prioritized them. Now, let us apply Eq. 1 to the prioritized test sequences to compute the value of APFD. From Table 4, it is observed that, m = number of faults = 9, n = number of test sequences = 7 and TF_i = number of faults detected. Putting the values of m, n, and TF_i (The position of the first test scenario in the ordering T′ of T that exposes fault *i*) in Eq. 1, we get,



```
0
      S0    S1    S2    S3    S4    S5    S6    S7    S8    S9
S0    0     1     0     0     0     0     0     0     0     0
S1    0     0     1     0     0     0     0     0     0     0
S2    0     0     0     1     0     1     0     0     0     0
S3    0     1     0     0     0     0     1     0     0     0
S4    0     0     1     0     0     1     0     0     0     0
S5    0     1     0     0     1     0     0     0     0     0
S6    0     0     0     1     0     1     0     1     0     0
S7    0     0     0     0     0     0     0     0     1     0
S8    0     0     0     0     0     0     0     0     0     1
S9    0     0     0     0     0     0     1     0     0     0

S0->S1->S2->S3->S1->


S2->S5->S1->


S3->S6->S3->
```

Fig. 5 Test sequences for the state machine diagram of Bank ATM systems given in Fig. 2

$$\mathrm{APFD} = 1 - ((1+3+2+2+5+3+1+5+4)/(7*9)) + 1/(2*7) = 0.6428571428571429$$

For the non-prioritized test sequences the APFD value is computed as follows:

$$\mathrm{APFD} = 1 - ((7+1+6+1+4+1+7+1+1)/(7*9)) + 1/(2*7) = 0.5952380952380953$$

We observed that the APFD value obtained for the *prioritized* sequences (using our approach) is more than the *non-prioritized* test sequences. Hence, for the given Bank ATM system, this approach achieves a higher APFD value than the other existing approaches. So, this approach increases effectiveness of the prioritized test sequences.

6 Comparison with related work

In this section, we describe the related research work in the area of UML-based testing. Most of the work on UML-based testing are based on state machines. Usman and Peng (2008) proposed an approach that used activity diagram for introducing mutation analysis. Their objective was to perform mutation analysis for verification and validation of applications based on activity diagrams. They defined mutation operator as syntactic errors according to the control flow and concurrency features. Pandey and Shrivastava (2011) proposed an integrated and costeffective test case prioritization approach to detect software faults during the maintenance phase. They considered three factors, Program Change Level (PCL), Test suite Change Level (TCL) and Test suite Size (TS). Chen et al. (2008) proposed an approach that used specification coverage to generate properties as well as design model to enable directed test generation using model checking. In their method, the number of test sequences is reduced and the maximum number of test sequences is bounded by the number of predicates in a state machine.

Kim et al. (2007) 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 were flattened and broadcast communications were eliminated. Then, data flow was identified by transforming the EFSMs into flow graphs, to which conventional data flow analysis techniques were applied.

Kalaji et al. (2009) proposed an approach in which an EFSM contained states, variables, and transitions among the states. EFSM of a class has an object state consisting of values assigned to data members. A transition has guard condition and action associated with the variables. A transition in the class diagram occurs as an external input. The transition takes place when the guard condition is slaked and the associated actions are executed. A CFG in UML state diagrams is identified in terms of the paths in the resulting EFSMs. A significant advantage of our approach over their approach (Kalaji et al. 2009) is that our approach generates test sequences by the use of the state machine and covers all independent sequences without redundancy.

Abdurazik and Offutt (2000) presented test criterion based on collaboration diagrams for dynamic testing and static checking. They adapted traditional data flow coverage criterion in the context of collaboration diagrams. It did not generate prioritized test cases. Gerhard (2005) presented a test sequence generation method from activity diagram which used condition classification tree. Flake and Muller (2003) presented a formal semantics for the dynamic behavior of UML models using the temporal OCL extension. Their approach performed semantic integration of UML statecharts into OCL language concepts by the formal definition of a statechart configuration. Gnesi et al. (2004) proposed a theoretical method for testing and conformance theories for UML state chart. A formal conformance testing relation was proposed for input-enabled transition systems with transitions labeled by input/output-pairs (IOLTSs). IOLTSs provided a suitable semantic model for a behavioral subset of state chart diagram. An automatic test case generation algorithm was proposed using UML state chart.

Latella and Massink (2001) proposed a formal testing framework for a behavioral subset of UML state chart diagrams (UMLSDs).

However, the results obtained so far are preliminary. The associated automatic test data generation procedures are difficult, and none of the reported results directly addresses specification-based software testing. We use the state machine diagram to extract test sequences to verify the behavior of the model. SMTSG algorithm is used to generate all possible valid test sequences for the state machine. After that, we calculate the effectiveness of test sequences using APFD metric based on fault detection capability.

7 Conclusion and future work

This paper presented a model based approach for test sequence generation using a state machine. The generated test sequences are prioritized based on fault detection capability. This experimental work in software testing has focused on generating test sequences using an algorithm called SMTSG. This work has described the effectiveness of test sequences using mutation faults. We have considered nine types of state faults for checking effectiveness of the generated test sequences. Our work also calculated the effectiveness of the prioritized test sequences using

Average Percentage Fault Detection (APFD) metric. In this experimental result, we have observed that the APFD value for the prioritized test sequences obtained using the proposed approach is 4.7619 % more than the non-prioritized test sequences. In future, we will prioritize test sequences using some heuristic algorithms.

References

Abdurazik A, Offutt J (2000) Using UML collaboration diagrams for static checking and test generation. In: 3rd International conference on the unified modeling language: advancing the standard. Springer-Verlag, Berlin, pp 383–395

Ammar HH, Yacoub SM, Ibrahim A (2001) A fault model for fault injection analysis of dynamic UML specifications. In: 12th IEEE international symposium on software reliability engineering, pp 383–395

Bernhard AK, Brandl H, Elisabeth J, Willibald K, Rupert S, Stefan T (2014) Killing strategies for model-based mutation testing. Softw Test Verif Reliab 24(4):1–32

Chen M, Mishra P, Kalita D (2008) Coverage-driven automatic test generation for UML activity diagrams. In: 18th ACM great lakes symposium on VLSI, ACM, pp 139–142

Elbaum S, Malishevsky AG, Rothermel G (2002) Test case prioritization: a family of empirical studies. IEEE Trans Softw Eng 28(2):159–182

Flake S, Muller W (2003) Formal semantics of static and temporal state-oriented OCL constraints. Softw Syst Model 2(3):164–186

Gerhard GH (2005) Component-based software testing with UML. Springer, Berlin

Gnesi S, Latella D, Massink M (2004) Formal test-case generation for UML statecharts. In: 9th IEEE international conference on engineering complex computer systems, pp 75–84

Gupta A, Jalote P (2008) An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. Int J Softw Tools Technol Transfer 10(2):145–160

Jorgensen PC (2008) Software testing: a Craftsman's approach. Auerbach Publication, Boca Raton

Kalaji A, Hierons RM, Swift S (2009) A search-based approach for automatic test generation from extended finite state machine (EFSM). In: IEEE testing: academic and industrial conference-practice and research techniques, pp 131–132

Kim SW, Clark J, McDermid JA (2001) Investigating the effectiveness of object-oriented testing strategies using the mutation method. Softw Test Verifi Reliab 11(4):207–225

Kim H, Kang S, Baik J, Ko I (2007) Test cases generation from UML activity diagrams. In: 8th IEEE ACIS international conference on software engineering, artificial intelligence, networking, and parallel/distributed computing, pp 556–561

Latella D, Massink M (2001) A formal testing framework for UML statechart diagrams behaviours from theory to automatic verification. In: 6th IEEE international symposium on high assurance systems engineering, pp 11–22

Pandey AK, Shrivastava V (2011) Early fault detection model using integrated and cost-effective test case prioritization. Int J Syst Assur Eng Manag 2(1):41–47

Punuganti SBA, Pattanaik PK, Prasad S, Mall R (2007) Model-based mutation testing of object-oriented programs. IT Bus Intell 49(2):1–9

Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27(10): 929–948

Shirole M, Kumar R (2013) UML behavioral model based test case generation: a survey. ACM SIGSOFT Softw Eng Notes 38(4): 1–13

Srivastava PR (2003) Putting your best tests forward. Softw Eng Best Practi IEEE 20(5):74–77

Srivastava PR (2008) Test case prioritization. J Theor Appl Inf Technol 4(3):178–181

Swain SK, Mohapatra DP, Mall R (2010) Test case generation based on state and activity models. J Object Technol 9(5):1–27

Usman F Peng LC (2008) Mutation analysis for the evaluation of AD models. In: international conference on computational intelligence for modelling control & automation, IEEE computer society, pp 296–301