Received December 9, 2018, accepted December 16, 2018, date of publication January 3, 2019, date of current version January 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2890791

Automatic Code Generation From UML State Chart Diagrams

SUNITHA E. V. AND PHILIP SAMUEL
Department of Computer Science, Cochin University of Science and Technology, Kochi 682022, India
Corresponding author: Sunitha E. V. (sunithaev@tistcochin.edu.in)

ABSTRACT The fact that event driven systems can be modeled and implemented using unified modeling language (UML) state chart diagrams has led to the development of code generation tools. These are tremendously helpful in making software system designs and can even generate skeletal source code from these designs. The implementation of such automatic code generation from state diagrams is not fully supported by the existing programming languages. The major down side is that there is no one-to-one correspondence between the elements in the state chart diagram and the programming constructs. The existing programming elements cannot effectively implement two main components of the state diagram namely, state hierarchy and concurrency. In this paper, we present a novel design pattern for the implementation of the state diagram which includes hierarchical, concurrent, and history states. The state transitions of parallel states are delegated to the composite state class. The architecture of the code generator and the step by step process of code generation from UML state machine are proposed in the paper. The proposed approach is implemented and compared with similar tools and the results are found to be promising.

INDEX TERMS Code generation, state machine, concurrency, hierarchy, history state, executable UML, MDD.

I. INTRODUCTION

Automatic code generation from system designs is an interesting research area in Model Driven Development (MDD) [34]–[38]. This concept has versatile dimensions in software industry. It is a part of model-based execution [10]. The idea behind this concept is that, before implementing a system, we can model it using standard notations, like UML [7], [8], [45]. Then compile and execute this model to test the working of the system even before its full implementation. This idea is quite interesting since the coding and testing phases of software development process are very much expensive. The concept of model compilation can reduce the effort we put for coding and testing, and in turn we can improve the quality of the software. It can reduce the bugs in the developed products. It helps us to refine the requirement specification.

Another important aspect of model based execution is the correlation between design and implementation code after some software maintenance. Normally, the system designs are used as a means of communication between the client and the developer. Those designs will be used for system implementation. During maintenance phase, the maintenance engineer changes the source code, but not the designs. So, each maintenance work reduces the correlation with the design and the source code. Gradually the system design becomes obsolete and it may not have any relation with the actual system. The model based execution gives a solution for this scenario. During maintenance the engineer can change the system models instead of the actual code, and then generate code out of the system model.

The concept of model execution comes from the idea of executable UML [10]. It says that a system can be modeled as communicating objects. Each object has its own state transition diagram. State diagram listen for events and act accordingly. So the system can be modeled using class diagrams, state chart diagrams and activity diagrams. The interesting part is that these diagrams can contribute much to the automatic code generation. The system modeled using these three diagrams can be translated to implementation code using code generation tools. Class diagrams help us to generate structural codes and the other two diagrams help us to generate behavioral codes. Out of these three diagrams, UML state chart diagrams are most popular standard for embedded system design [46] and event driven system modeling [33], [40], [43]. This paper concentrates on state machines since it can generate 100% code out of simple state chart diagrams.

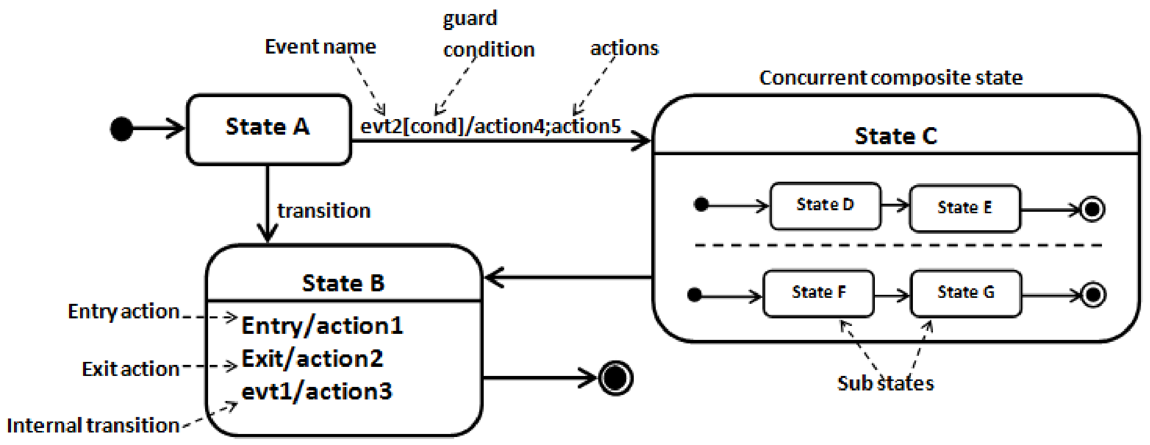


FIGURE 1. State chart essentials (Source: UML Reference manual, p. 527).

UML state chart diagrams can effectively represent the behavior of event driven systems aka reactive systems. The behavior of the event driven systems changes with the interactions (events) by the environment. The state diagrams show that the behavior of a system depends on the current input to the system as well as the previous interactions by the environment. Event driven systems modeled using the state machines can represent the full life cycle of an object. The different states of the object and the transition between those states are all portrayed in this. The challenge is to work out an efficient method to convert state charts to a program since there is no programming construct exists to directly represent elements in the state diagram.

In this paper we present a method to convert hierarchical states, concurrent and history states to Java code. In our method, we follow a design pattern based approach. A design pattern gives the overall implementation outline using a class diagram [41], [42]. The surveys on code generation from state machines [26] show that the research outcomes are not giving an effective method to implement the concurrent states. We present a design pattern to implement the state hierarchy, concurrency, and history state.

The main contributions of the paper are as follows:

- It presents an easily understandable and reusable design pattern for state machine implementation.
- The design pattern is expandable and is able to handle hierarchical states.
- It gives an effective method to implement compo-site states with parallel regions in object oriented way.
- It presents a simple method to keep shallow and deep history in a state machine

The paper is organized as follows. Section 2 gives an overview of the UML state chart diagrams. Section 3 surveys different approaches to program UML state charts. Section 4 presents the concept of implementing hierarchy, concurrency, and history in state machines. Section 5 introduces a design pattern for implementing UML state chart diagrams. Section 6 presents a case study to demonstrate the proposed approach. Section 7 describes the code generation process. Section 8 presents the related works. Section 9 evaluates and compares the proposed approach with related works and Section 10 concludes the paper.

II. OVERVIEW OF UML STATE CHART DIAGRAMS

A state machine can be defined as a graph of states and transitions [7], [44]. State chart diagram may be attached to classes, use cases, and collaborations to describe the dynamics of an individual object. It models all possible life histories of an object of a class. Any external influence to the object is called as an event. The response to the event may include the execution of an action and transition to a new state. Events may have parameters that characterize each individual event instance. Inheritance and concurrency can be modeled in state machines. A sample state chart diagram is shown in the Fig.1.

The commonly used features of a state chart diagram are listed as state, transition, event, guard, entry action(s), exit action(s), transition action(s), and internal action inside state.

The state of an object is defined as a time period in its life. During this period, the object may wait for some event, or it may perform some activity. There can be named states as well as unnamed (anonymous) states. The transition is the response of the object to an event occurrence. Transition will have an event trigger and a target state. It may include a guard condition, and an action. There can be different types of transitions like, entry action, exit action, external transition, and internal transition. A Guard condition is a Boolean expression which is evaluated when a trigger event occurs. If the expression evaluates to true, then the transitions occurs. An action is an atomic computation which can be a simple assignment or arithmetic evaluation statement. It can also be a sequence of simple actions. The Entry and Exit actions exist in composite states which contains nested states. Entering the target state executes an entry action and when the transition leaves the original state; its exit action is executed before the action on the transition and the entry action on the new state.

The composite state may contain sequential states (OR type states), or (and) concurrent states (AND type states). The concurrent states form orthogonal regions in the composite states. The states in two orthogonal regions are concurrent states. For example, in figure 1, state C is a composite state which contains two orthogonal regions. In one region, there are two sequential states D and E, and in the other region there are two states F and G. The existence and execution of state F and G is independent of that of states D and E. In other words, states F and G are concurrent with states D and E.

In this paper, we discuss how these elements can be represented in an object oriented program. We assume that the source code has to be generated automatically from the state chart diagram with the help of some CASE tool. The following section examines different methods to map the state chart diagram to program constructs.

III. COMMON APPROACHES TO IMPLEMENT STATE MACHINES

In this section we discuss standard state machine implementation techniques which we can find in the literature. The methods include nested switch statement, state table, and state design patterns. The section ends with a proposal of new approach which supports hierarchical state machines and concurrent states.

A. USING SWITCH STATEMENT

The finite state diagram shown above describes the behavior of a combination lock whose combination is 2-1-8. The Finite State Machine (FSM) can be in one of four possible states: NoneRight, OneRight, TwoRight, and Open.

The general structure of the state chart implementation using switch statement is shown below.

```
void LockStateMachineCASE( unsigned char NewEvent)
{static unsigned char CurrentState;
unsigned char NextState;
switch(CurrentState)
{
case NoneRight:
switch(NewEvent)
{case '2':NextState = OneRight;
break;
default: break;
}
break;
case OneRight:
switch(NewEvent)
{case '1':NextState = TwoRight;
break;
default: NextState = NoneRight;
break;
}
break;
case TwoRight:
switch(NewEvent)
{case '8':NextState = Open;
OpenLock();
break;
default:NextState = NoneRight;
break;
}
break;
case Open: NextState = NoneRight;
    LatchLock();
    break;
}
CurrentState = NextState;
return;
}
```

The system state is implemented as a variable and events are implemented as methods. The switch statement receives the current state and the nested switch statement chooses appropriate action for each event. This is straight forward method for state chart implementation [5]. The entire system will be represented in a class called context class and the event methods are its members.

Even though it's a simple method of state chart implementation, it can't support concurrent states in a state chart diagram. In addition to that, the composite states cannot be implemented using this method, since the state hierarchies cannot be represented in switch case statements.

A different implementation method is proposed by Jakimi and Elkoutbi [2]. In his approach, the state machine is represented as a class and the states are the attributes of the class. The events in the system are represented as the member functions of the class. An example of this approach is given in figure 2. It is a state diagram of an engine which has two states; idle and running. One event in the system is switchON which causes the state transition. The state diagram is implemented as class Engine. An integer attribute, on, is defined to represent the system state. When the system in idle state, on = 0; and when the system is in running state, on = 1. The event is represented as the member function switchOn() which changes the value of state variable.

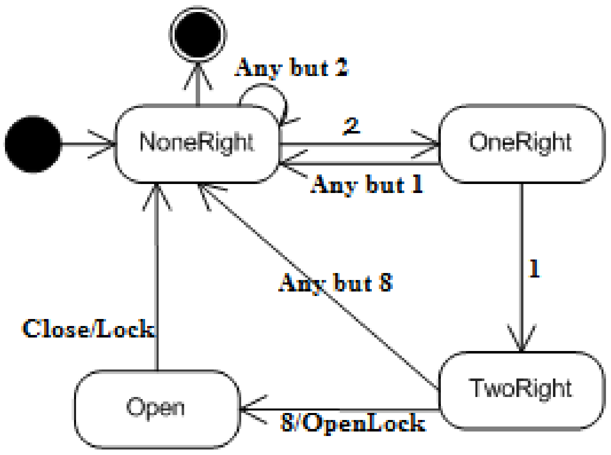


FIGURE 2. State machine for a combination lock.

State\events	2	1	8	Any but 1	Any but 2	Any but 8	Close
NoneRight	OneRight				NoneRight		
OneRight		TwoRight		NoneRight			
TwoRight			Open openLock()			NoneRight	
Open							NoneRight lock()

FIGURE 3. State table structure for state diagram in figure 2.

B. USING STATE TABLES

Another method for representing state machine is state tables. It is a two dimensional structure like a matrix. Each row represents different states of the system and the columns show the possible events that can happen in the system. Each element in the table shows which action has to be taken when an event occurs and the next state of the system as shown in figure 3. This approach is more convenient for coding simple state chart diagrams and better than switch case method. Editing the switch statements will be little bit confusing when the number of states and events increase. In nested switch method, we should have n number of cases, and inside each case we should have m number of nested case statements, where n and m are the number of states and events respectively. In the case of state tables, editing the table will be easier than switches due to its 2D arrangement.

As the number of states and events increases the table size increases drastically. It is the main drawback of this approach. Moreover the table size does not depend on the number of transitions. Hence the table can be large even though the number of transitions is small. This in turn results in wastage of memory. Moreover, concurrent states cannot be implemented in this method.

C. USING STATE DESIGN PATTERNS

In state design pattern approach, there will be a class diagram pattern that has to be followed for implementing all state chart diagrams [6], [9]. There will be one class in the pattern which represents the context (domain) of the state chart diagram. The states in the state chart diagram are abstracted in a single abstract class which will act as an interface to the states in the state chart. The events will be the virtual member functions of the abstract state class. Each individual state in the state chart will be represented as the object of the derived class of abstract state class. If there are 'm' states in the state chart, then there will be 'm' different concrete state classes derived from the abstract state class. A sample state design pattern is shown in Fig. 4.

The object of the context class represents the domain object that needs to be represented in the program. The context class will have a data member (state variable) which represents the current state of the domain object. All the events are represented as member functions of the context class which in turn delegates the function to the corresponding state class objects.

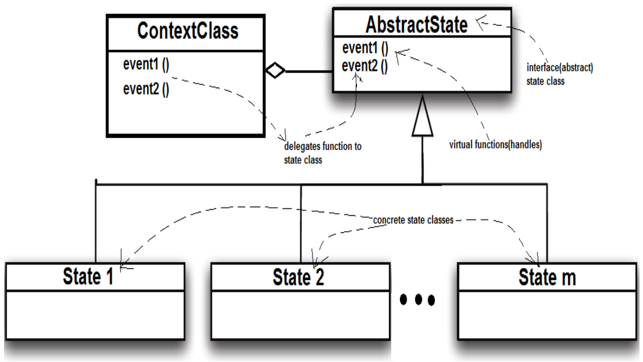


FIGURE 4. Sample state design pattern.

Using state design patterns we can bring the object orientation in the state machine implementation. The domain object, whose state chart is drawn, is implemented as the object of the context class, each state of the domain object is implemented as the object of the corresponding concrete state class. Events are represented as the handles of the abstract state class and the transitions are accomplished by updating the state variables. This approach supports code reusability and avoids redundancy in coding.

There can be variable type of patterns that can be used to represent the state chart diagram. Fig. 5 and Fig. 6 other two sample patterns. In both the patterns, there is an abstract class which acts as an interface for the state classes. The interface will be connected to the context class. The pattern, shown in Fig. 6, has an additional object called collaboration object to accomplish the sub states. It is an abstract class which acts as an interface for the sub states [3].

This object oriented approach creates some inconvenience too. In order to add a new state, we have to derive one more

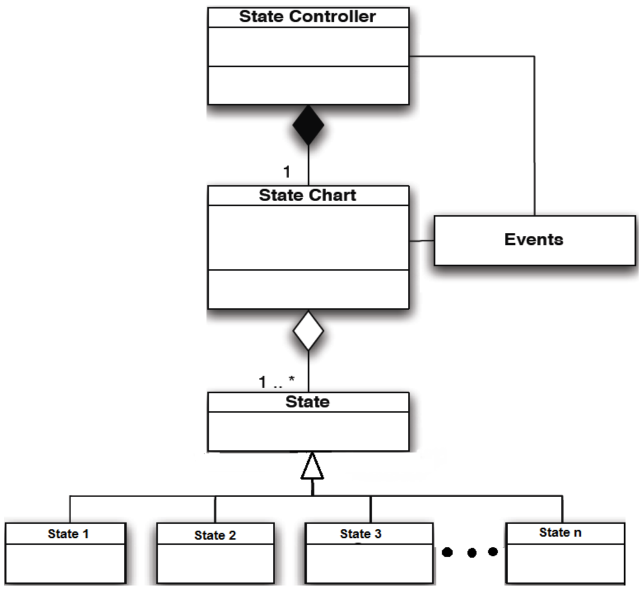


FIGURE 5. structure of state chart implementation.

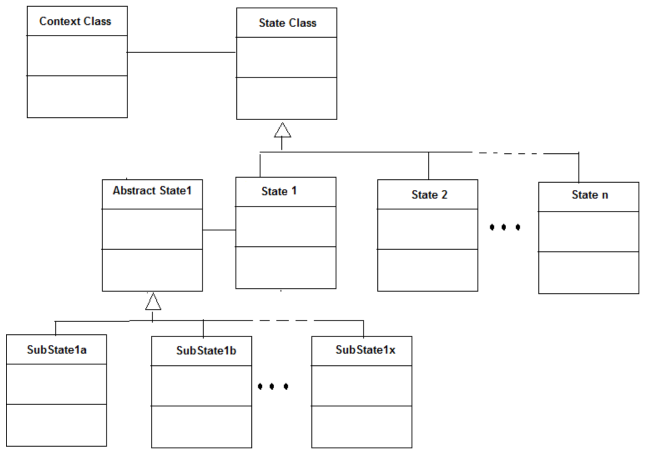


FIGURE 6. Implementation structure of state charts using collaborator object.

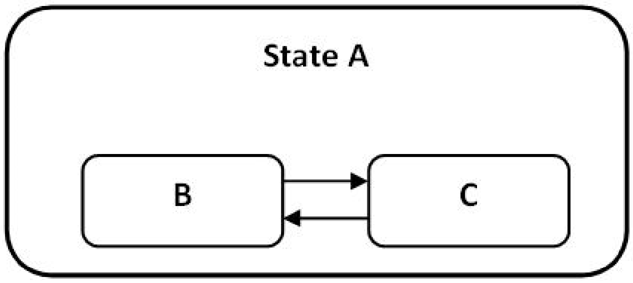


FIGURE 7. Composite state A with two sub states.

concrete state class from the abstract state class. Similarly, to add a new event, we need to add one more virtual function to the abstract state class.

The above mentioned methods failed to represent the concurrent states. Some literature, based on State Design Patterns, attempted to address this issue but resulted in very complex design patterns [24], [27], and failed to implement the key features of state machine. For example, in Spinke [24], it requires 17 classes in the implementation for representing a state machine with 6 states and 6 events. It makes the implementation very bulky.

IV. IMPLEMENTING HIERARCHICAL, CONCURRENT AND HISTORY STATES

In design pattern based approach, each state of the system (or object) is implemented as a class. If the object has three states, then there will be 3 classes, representing each state, in the implementation. Generalization is applied when there are hierarchical states (composite states). For example in Fig 7, state A is a composite state. It contains two sub states B and C. so there will be three classes A, B and C. In the implementation pattern they appear as in Figure 8. The sub states B and C share the properties of the super state A. So, inheritance (generalization) is the best choice to implement the state hierarchy.

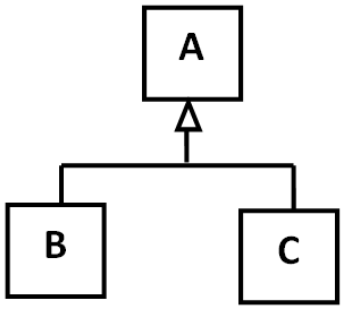


FIGURE 8. Implementation Pattern of the composite state A.

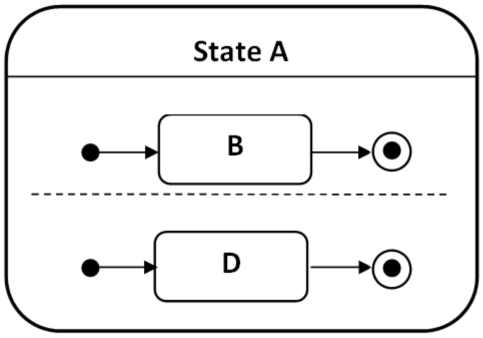


FIGURE 9. Composite state A with two orthogonal regions.

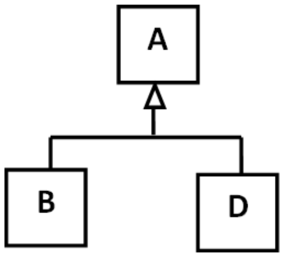


FIGURE 10. Implementation Pattern of the composite state A with orthogonal regions.

There can be composite states with orthogonal regions. For example, in Figure 9 we can see, state A contains 2 sub states B and D. So the implementation contains three state classes, class A, class B and class D. The super state and sub states are implemented using generalization as in Fig 10. This generalization can only show that A is a super state and B and D are the sub states. Another important property of the sub states is not addressed here. It is the concurrency between the sub states. State B belongs to one region and state D belongs to another region. That means the existence of state B is independent of that of state D. The state transitions of the state in the orthogonal regions are independent of each other, or in other words, they are concurrent.

To implement the concurrent states, we defined a base class called OrthogonalProperty. According to our new approach, every composite state class has to inherit the properties of the OrthogonalProperty class as in Figure 11. The OrthogonalProperty class has two important attributes to store the

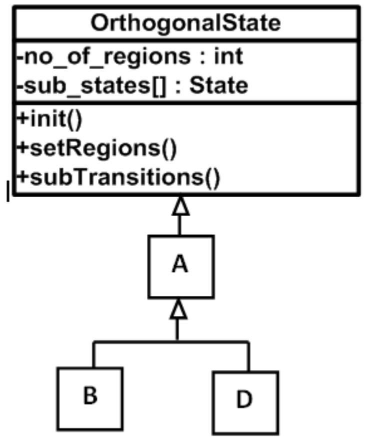


FIGURE 11. Implementation Pattern of the composite state A with orthogonal regions.

number of regions as well as the active sub states. The number of orthogonal regions in the composite state will be stored in the attribute no_of_regions. The active sub states will be registered in the sub state array named as sub_states[]. If there are two orthogonal regions, then there can be a maximum of two active sub states. As the number of regions increases, the entries in the sub state array increases. Now, the sub state transitions are implemented based on the sub state array. Whenever there is a transition between sub states, the sub state array will be updated with the new target sub state.

Next task is to implement history states. There can be two types of history; shallow and deep. Deep history gives the inner (nested) states that were active previously, and the shallow history gives the outer state which was active previously. According to the above pattern for orthogonal state, the active state (outer state) is managed by context class and the nested states are managed by the composite class. So it is easy to maintain the shallow history in Context Class and the deep history in CompositeState class.

V. DESIGN PATTERN FOR HIERARCHICAL, CONCURRENT AND HISTORY STATES

In this section, we present the design pattern for implementing the UML state chart diagram. In our previous work [39] we have presented a design pattern for concurrent and hierarchical states. Another important feature of state diagram, the state history, is incorporated in the new pattern, named as "Template HHCStateMachine", as shown in figure 12. As per Gamma [47] giving name to a design pattern will help us to refer to the pattern frequently. The pattern gives a simple and easy to use method for state chart implementation using object oriented concepts.

The states and events in a system are implemented as classes in the pattern. Every state machine can have an active state. There will be events in the system that may or may not change the state of the system. State changing events should initiate state transition. All these terms are included in the StateMachine class which is a blue print of every state machine.

The ContextClass is the class which represents the actual system to be implemented. It has an even dispatch function. This method is used to delegate the events to the corresponding state classes.

State hierarchy is represented using inheritance of state classes. The pattern defines an abstract class called State which acts as the base class for deriving the states in the system. Each state in the system is defined as a derived class of State class. If there is a composite state, all the sub states will be implemented as the derived class of the composite state class. It keeps the semantics of composite state in UML state chart diagram. According to UML, the sub states have

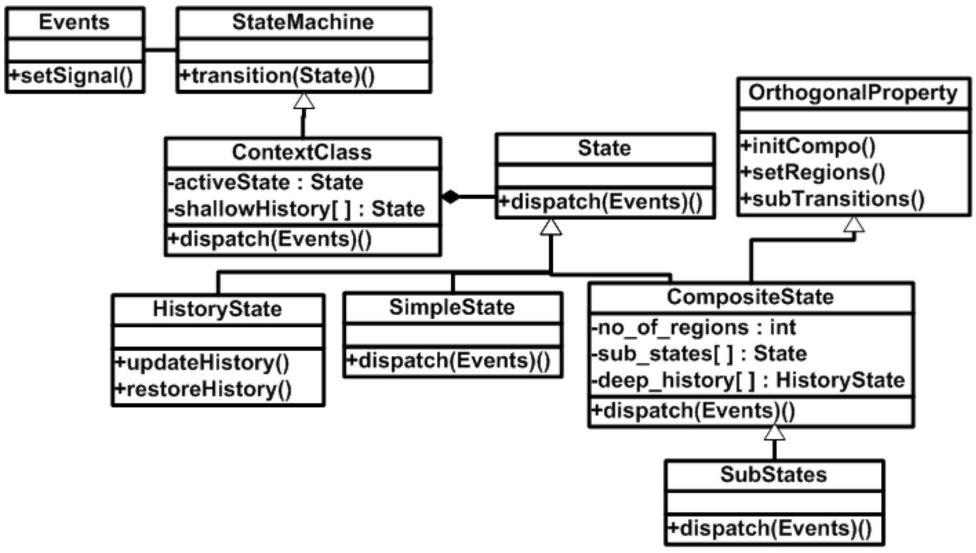


FIGURE 12. The proposed design pattern, "Template HHCStateMachine".

the properties of the composite (outer) state. This property can be satisfied by the use of inheritance to derive the sub state classes.

Another important feature of the state chart diagram is parallel regions inside the composite states. The pattern defines a special class called OrthogonalProperty which captures the features of parallel regions in the composite state. It sets the number of regions and the transitions between sub states using the methods setRegions() and subTransitions() respectively.

The CompositeState class maintains the properties of the composite states with or without parallel regions. It has three main attributes; no_of_regions, sub_states[ ], and deep_history[ ]. In a composite state there can be one or more regions. It is stored in the first attribute. If there are multiple regions in a composite state, there will be parallel states. These states which are active simultaneously are stored in the second attribute. Before state transition, the old state is stored in the third attribute. Hence, concurrent states and history are maintained using the OrthogonalProperty class. The deep_history[ ] stores the inner states which were active previously. Shallow history is kept as an attribute of the StateMachine class. The transition between sub states is implemented using the method subTransitions().

The mapping of the state machine elements and the Object Oriented Programming constructs is shown in table 1.

TABLE 1. Mapping State machine elements to program constructs.

State Machine Element	Program Construct
State	State Class
Transition	method in StateMachine class
Event	Events class
Entry / Exit Actions	method in State class
Internal Action	method in State class
Hierarchical States	hierarchy of State Classes
Concurrent Transitions	method in the OrthogonalProperty class
Shallow History	attribute in StateMachine class
Deep History	History State Class

The skeletal code structure of the pattern is as follows. It includes the classes for Events, State, StateMachine, ContextClass, OrthogonalProperty, and CompositeState.

```
public class Events {
  public void setSignal(){   }
}
public class State {
  public void dispatch(ContextClass cc, Events e){   }
}
public class StateMachine{
  public void transition(State target){   }
}
Fig. 13. UML state diagram representing the alarm clock
public class ContextClass extends StateMachine {
     State activeState;
     State shallowHistory;
     public ContextClass () {   }
     public void init(){   }
     public void dispatch(Events e) {    }
}
public interface OrthogonalProperty {
     public void init();
     public void subTransition(int region, State target);
     public void setRegions(int no_of_regions );
}
public class HistoryState extends State
{   void restoreHistory(){. . .}
     public void updateHistory(){. . . . .}
}
public class CompositeState extends State implements
OrthogonalProperty{   int no_of_regions;
     State[] sub_states; State[] deep_history;
   public void init(){   }
      public void subTransition(int region, State target)
{. . . . . . . . . . . .  }
      public void dispatch(ConetxtClass context, Events e)
{. . . . . . . . . . . . }
   public void setRegions(int no_of_regions) {
          . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .     }
}
```

VI. CASE STUDY

In this section we present two case studies to show the implementation of the pattern "Template HHCStateMachine".

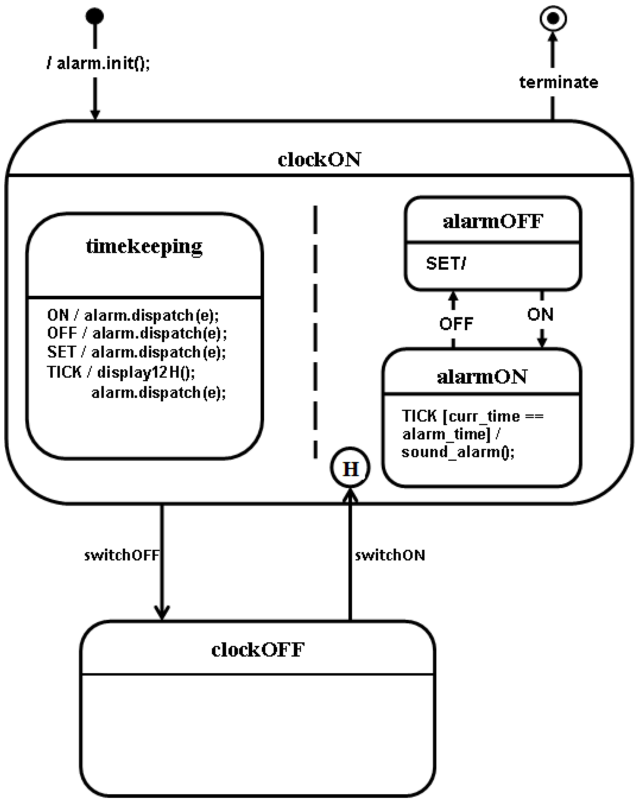


FIGURE 13. UML state diagram representing the alarm clock.

TABLE 2. State transition table of the alarm clock.

Current State	Sub State	Events	[guard]	next state
clockON	timekeeping	TICK		timekeeping
		SWITCH_OFF		clockOFF
	alarmON	TICK	curr_time = alarm_time	alarmON
		SWITCH_OFF		clockOFF
		ALARM_SET		alarmON
		ALARM_OFF		alarmOFF
	alarmOFF	ALARM_SET		alarmOFF
		ALARM_ON		alarmON
		SWITCH_OFF		clockOFF
clockOFF		SWITCH_ON		clockON

First one is an alarm clock and the second one is a microwave oven.

A. IMPLEMENTING ALARM CLOCK

We consider the case of an alarm clock. The clock can be in ON state or OFF state. When the clock is ON it can be in two modes simultaneously, timekeeping mode and alarm mode. There are six events in the alarm clock; TICK, ALARM_ON, ALARM_OFF, SWITCH_ON, SWITCH_OFF and SET. TICK is the advancement of time in seconds. ALARM_ON is to switch on the alarm, and ALARM_OFF is to switch off the alarm. SET signal is used to set the alarm time. SWITCH_ON and SWITCH_OFF signals are used to switch on and switch off the clock respectively. When clock is off, the SWITCH_ON event changes the clock state to clockON state. In the ON state, by default, the clock will be in time-keeping and alarmOFF state. The state changes are shown in Table 2. The state diagram of the alarm clock is shown in Fig.13. The implementation pattern of the Alarm clock is shown in Figure 14.

The implementation of the AlarmClock has 6 state classes. The context class here is the AlarmClock. It uses the Events class and the State class for setting the state of the system and event dispatching. The initial state is set to TimeKeepingState and AlarmOff. Whenever an event encounters the corresponding event handling function will be called by using the run time polymorphism. Whenever the system enters the composite states, the init() function of the class has to be invoked. So this function call is included in the transition function

```
public class AlarmClock extends StateMachine {
…………………………………………..
…………………………………………..
…………………………………………..
public static ClockOnCompoState clockOn = new Clock-
OnCompoState();
   public static ClockOff clockOff = new ClockOff();
   public AlarmClock(int hr, int min, int sec)
   {   curr_time_hr = hr;
        curr_time_min = min;      curr_time_sec = sec;    }
    final public void init(){      m_state = clockOff;
          tran(m_state);
}
    final public void dispatch(ClockEvents e)
    {              m_state.dispatch(this, e);      }
    final public void tran(ClockState target){
          shallowHistory = m_state;
          m_state = target;
          if(m_state == clockOn)
          {   flag++;
             if(flag == 1)           {   clockOn.initCompo();}
             else                  {clockOn.restoreHistory();}
          }}
               ……………………………………….
}
```

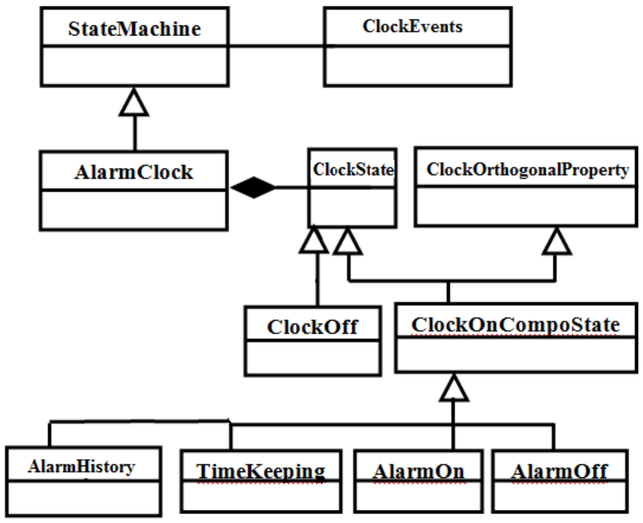


FIGURE 14. Implementation Pattern for alarm clock.

The clockOn state is a composite state. It contains two parallel regions; one for time keeping and the other for alarm. It is implemented as ClockOnCompoState. It also implements the sub transition function and history restoring function along with the event dispatch function.

```
public class ClockOnCompoState extends ClockState implements ClockOrthogonalProperty{
.................................................
.................................................
  public void initCompo()
  {
      subTransition(1,timing);
      subTransition(2,alarmoff);
  }
  public void subTransition(int region, ClockState target)
  {
          sub_states[region] = target;
                    ah.updateHistory(this);
  }
   public void dispatch(AlarmClock context, ClockEvents e)
 {
      int i = 0;
      for(i = 1;i <= no_of_regions;i++) {
        sub_states[i].dispatch(context, e);
      }    }
  public void restoreHistory()
  {          hs.restoreHistory(this);   }
 }
```

The clockOff state receives only SWITCH_ON event which causes state transition to clockOn state. It is implemented in the class ClockOff.

```
public class ClockOff extends ClockState{
     public void dispatch(AlarmClock context, ClockEvents e){
       switch(e.signal){
         case SWITCH_ON:
 {context.tran(AlarmClock.clockOn);break;}
       } } }
```

During alarmOff state, the clock receives ALARM_SET, ALARM_ON, and SWITCH_OFF events. It is implemented as AlarmOffState class. The alarmOn state is implemented as AlarmOnState class. It accepts TICK, ALARM_SET, ALARM_OFF, and SWITCH_OFF events.

ALARM_ON causes sub transition to alarmOn state. During TICK event, the current time is matched with alarm time and if matches it generates alarm sound. ALARM_SET event prompt the user to enter the alarm time. ALARM_OFF event causes sub transition from alarmOn state to alarmOff state.

```
public class AlarmOffState extends ClockOnCompoState{
    Scanner sc = new Scanner(System.in);
     public void dispatch(AlarmClock context, ClockEvents e){
      switch(e.signal){
      case ALARM_SET      :
    {System.out.println("ENter Hr: ");
                  context.alarm_time_hr = sc.nextInt();
                   System.out.println("ENter Min: ");
                  context.alarm_time_min = sc.nextInt();
                  System.out.println("ENter Sec: ");
                  context.alarm_time_sec = sc.nextInt();
                  System.out.println("Alarm Set to —> "+context.alarm_time_hr+":"+context.alarm_time_min+ ":"+context.alarm_time_sec);
                break;}
  case ALARM_ON       :  {
                  super.subTransition(2, alarmon);
                break;}
  case SWITCH_OFF:  {
    context.tran(AlarmClock.clockOff);break;
  }    }    }  }
public class AlarmOnState extends ClockOnCompoState{
  Scanner sc = new Scanner(System.in);
   public void dispatch(AlarmClock context, ClockEvents e){
       switch(e.signal){
           case TICK     : {
if((context.curr_time_hr == context.alarm_time_hr)&&
(context.curr_time_min == context.alarm_time_min)&&
(context.curr_time_sec == context.alarm_time_sec))
           {java.awt.Toolkit.getDefaultToolkit().beep();}
                 break;}
   case ALARM_SET      :  { System.out.println("ENter Hr: ");
                  context.alarm_time_hr = sc.nextInt();
                   System.out.println("Enter Min: ");
                  context.alarm_time_min = sc.nextInt();
                   System.out.println("Enter Sec: ");
                  context.alarm_time_sec = sc.nextInt();
                   break;}
   case ALARM_OFF      :  {    super.subTransition(2, alarmoff);break;          }
   case SWITCH_OFF     :                   {
context.tran(AlarmClock.clockOff);break;
  }    }    }  }
```

The timekeeping state is implemented as TimeKeepingState class. This state is a sub state of the clockOn state. So, the TimeKeepingState is inherited from ClockOnCompoState. This class handles two events; TICK event and SWITCH_OFF event. TICK event advances the clock time by one second. SWITCH_OFF event causes the state transition to clockOff state.

```
public class TimeKeepingState extends ClockOnCompoState{
  @Override
     public void dispatch(AlarmClock context, ClockEvents e){
       switch(ClockEvents.signal){
           case TICK     : {AlarmClock.curr_time_sec++;
                 if(AlarmClock.curr_time_sec == 60)
                 {AlarmClock.curr_time_min++;
                   AlarmClock.curr_time_sec = 0;
                   if(AlarmClock.curr_time_min == 60)
                   {  AlarmClock.curr_time_hr++;
                     AlarmClock.curr_time_min = 0;
```

```
                    if(AlarmClock.curr_time_hr == 13)
                     {   AlarmClock.curr_time_hr = 0;
                     }       }   }       break;}
        case SWITCH_OFF:  {context.tran(AlarmClock.clock
Off);
                    break; } } } }
```

The history state is implemented as HistoryState class includes a restoreHistory() function which restores the previous state of the Alarm Clock.

```
  public class AlarmHistory extends ClockOnCompoState {
  .........................
  public void updateHistory (ClockOnCompoState compo()
  {deepHistory = compo.sub_states[2];}
    public void restoreHistory(ClockOnCompoState compo)
    {   compo.sub_states[2] = deepHistory;      }
  }
```

B. IMPLEMENTING MICROWAVE OVEN

In this section we consider another case study, a microwave oven, figure 15. It includes parallel regions, history states and composite states. So it is a perfect case study to explore the proposed pattern "Template HHCStateMachine".

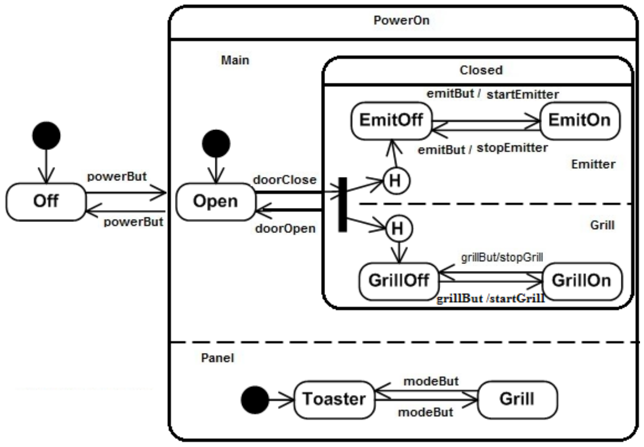


FIGURE 15. UML state diagram representing the microwave oven.

The microwave oven has mainly two states, 'Off' and 'PowerOn' states. PowerOn is a composite state with two parallel regions; 'Main' and 'Panel'. In 'Main' region we have two states door Open and Closed states. Again Closed state is a composite state with two parallel regions; Emitter and Grill. History states are attached to each region. The region Emitter includes two OR states; EmitOn and EmitOff. The Grill region has two OR states GrillOn and GrillOff. The 'Panel' region has two OR states; Toaster and Grill.

There are six events that can be occurred in the microwave oven. Pressing power button, emit button, grill button, and mode button are the four events. Open and close the oven door are the rest two events.

Initially the system is in Off state. When the power button is pressed, the system changes to PowerOn state. If the power button is again pressed, it will be switched off, that is, resume to Off state.

In PowerOn state, initially the system is in door open state as well as in Toaster mode. When the mode button is pressed, the Toaster mode will change to Grill mode. If the mode button is again pressed, the system will resume to the Toaster mode. When the door is closed, the system changes to the Closed state and the Emitter and the Grill will be initialized. When we first close the door, the emitter and grill will be in off state; EmitOff and GrillOff. When we resume to the Closed state again, then it will retain the previously active states; EmitOff/EmitOn and GrillOff/GrillOn. Pressing power button at any state will result in the transition to Off state.

The total number of states in the system is twelve, including the history states. So the implementation of this system, the microwave oven, will have twelve state classes, each represents the individual states of the system. In addition to this, there will be five additional classes. One class to represent the system itself, the context class MicroWaveOven, and another one to represent the events in the system, MWEvents. Other three are the common base classes, StateMachine, MWStates, and MWOrthogonalProperty. The class diagram for the implementation of the microwave oven is shown in figure 16. Some of the implemented classes have been shown here.

```
  public class MicroWaveOven extends StateMachine {
  ...........................
  ...........................
    final public void init(){m_state = off;tran(m_state);}
      final public void dispatch(MWEvents e)
    {       m_state.dispatch(this, e);   }
    final public void tran(MWState target){
        shallowHistory = m_state; m_state = target;
            if(target == poweron){
                        System.out.println("POWER ON");
                        poweron.initCompo();
                        }   }
  }
  public class MWEvents {
  .....................................
  .....................................
  public void setSignal(int i){       switch(i){case 1: signal
= MWEvents.events.POWER_BUT;break;
  case 2: signal = MWEvents.events.DOOR_CLOSE;break;
  case 3: Signal = MWEvents.events.DOOR_OPEN;break;
  case 4: signal = MWEvents.events.EMIT_BUT;break;
  case 5: signal = MWEvents.events.GRILL_BUT;break;
  case 6: signal = MWEvents.events.MODE_BUT;break;
    }  }}
  public   class   PowerOnCompoState   extends   MWState
implements MWOrthogonalState{
  ....................................
  ....................................
    public void initCompo()
    {   subTransition(1,open);subTransition(2,toaster);   }
```

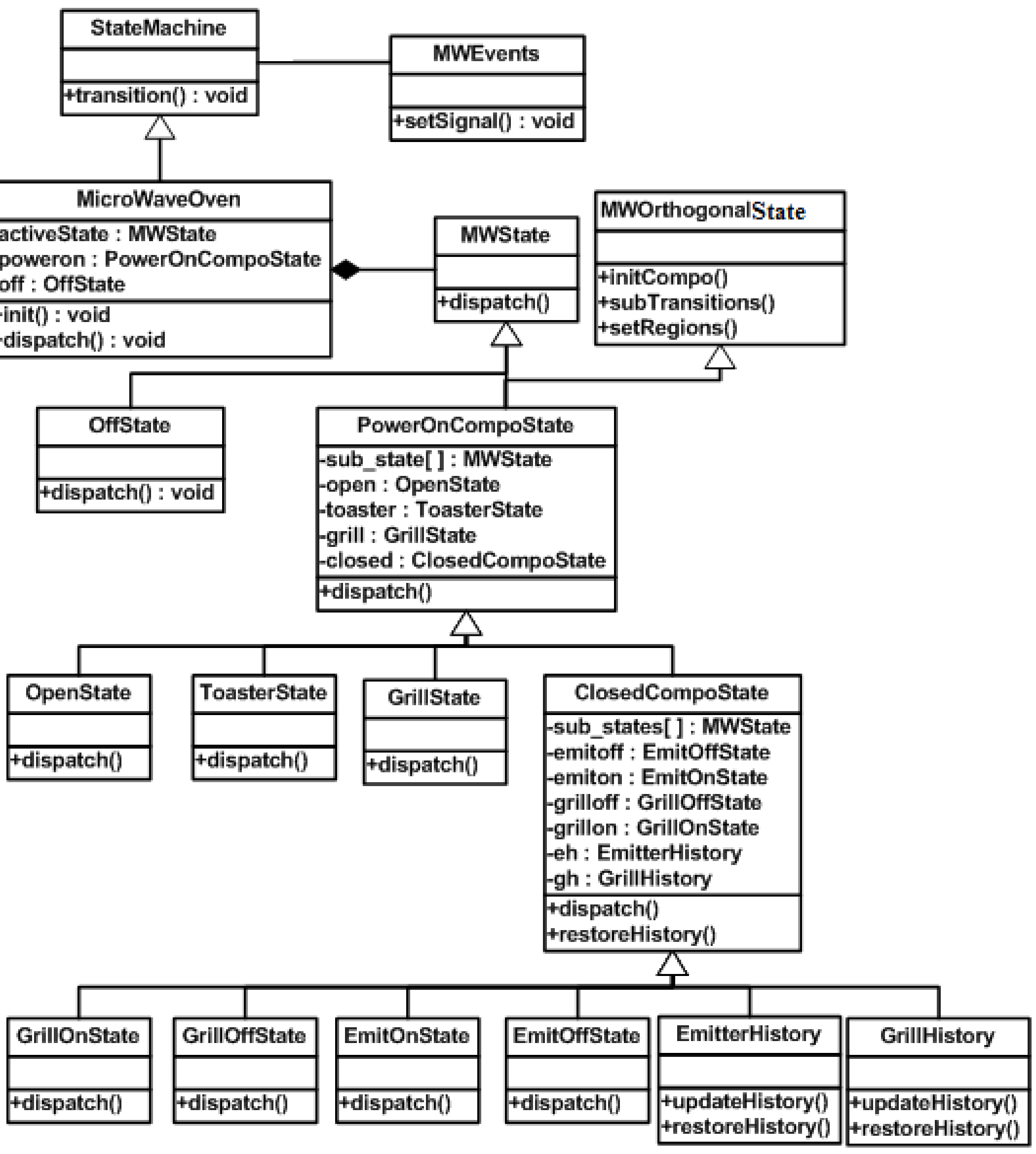


FIGURE 16. Implementation Pattern for the microwave oven.

```
    public void subTransition(int region, MWState target)
    {sub_states[region] = target; if(target == closed){
  if(flag == 0) {closed.initCompo(); flag++; }
            else{closed.restoreHistory();}}
            } }
        public void dispatch(MicroWaveOven context,
MWEvents e){.....................
        for(i = 1;i <= no_of_regions;i++){
          sub_states[i].dispatch(context, e);
        }     }
      public void setRegion(int r) {
  ...........................    }}
  public class ClosedCompoState extends PowerOnCom-
poState {
  ........................
```

```
  ........................
  public void subTransition(int region, MWState target)
  {sub_states[region] = target; eh.updateHistory(this);
              gh.updateHistory(this);     }
  public    void    dispatch(MicroWaveOven    context,
MWEvents e)
  {..................
  for(i = 1;i <= no_of_regions;i++){sub_states[i].dispatch
(context, e);     }       }
  public void restoreHistory(){eh.restoreHistory(this); gh.
restoreHistory(this);  }
  ..................................
  ..................................
  }
  public class Toaster extends PowerOnCompoState{
```

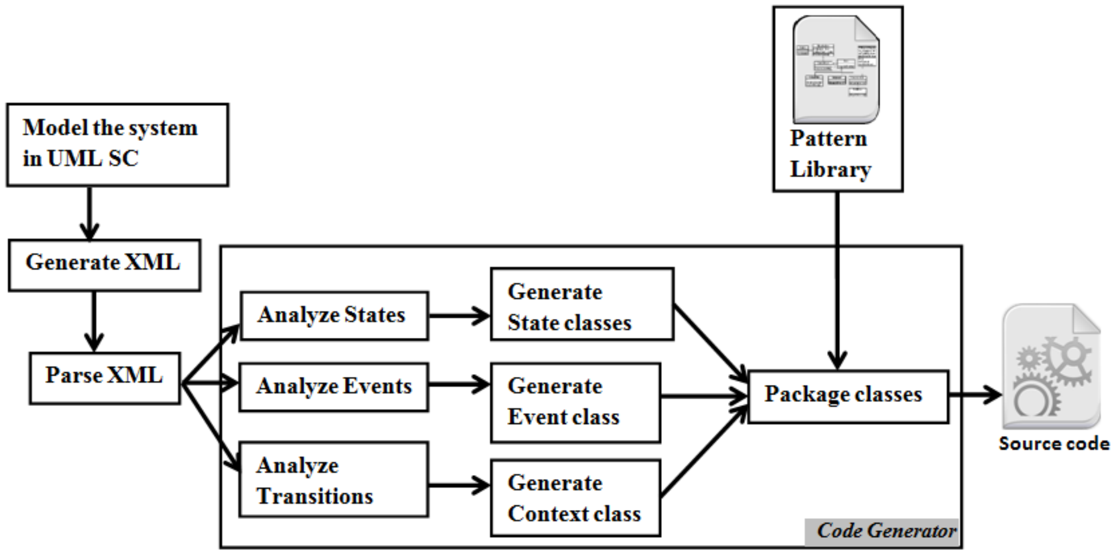


FIGURE 17. Architecture of Code Generator.

```
        public void dispatch(MicroWaveOven context,
MWEvents e){
    switch(e.signal){
    case MODE_BUT: {this.subTransition(2,grill);break;}
    case POWER_BUT: {context.tran(MicroWave.off); break;
}}  }
    }
    public class EmitterHistory extends ClosedCompoState{
    .....................................
      public void updateHistory(ClosedCompoState compo)
      {         deepHistory = compo.sub_states[1];   }
      public void restoreHistory(ClosedCompoState compo)
      {     compo.sub_states[1] = deepHistory;          }
    }
```

VII. THE CODE GENERATION PROCESS

The proposed pattern is saved in the pattern library. This pattern library is used during code generation. The code generation process includes the system modeling in UML state chart diagram, generation of XML for the model, parsing XML and then generating code. The process of code generation is depicted in figure 17.

The representation of the state chart diagram is based on State Chart extensible Markup Language (SCXML) [50]. Based on SCXML, the tags <state>, <transition>, <onentry>, <onexit>, <initial>, . . . <final>, <parallel>, and <history> are used to represent the states, transitions, entry condition, exit condition, initial state, final state, parallel states, and history states respectively. For example, a state with two transitions is given below.

```
<state id = s">
    <transition event = “e” cond = “x == 1” target =
“s1”/>
    <transition event = “e” target = “s2”/>
</state>
```

In state ‘s’, if an event ‘e’ occurs and the condition ‘x == 1’ is satisfied the state will take transition to state ‘s1’. If the condition is not satisfied, the state will take a transition to the state ‘s2’.

The parallel states are represented as follows.

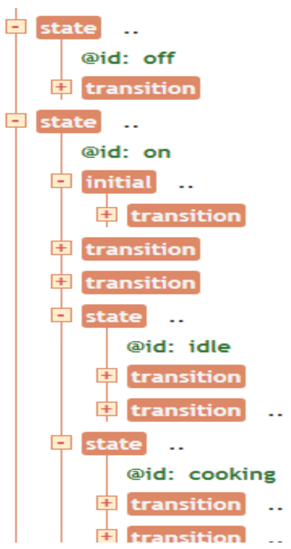
```
<parallel id = “p” >
<transition event = “done.state.p” target =
  <state id = “S1” initial = “S11”>
        ......................
        ......................
  </state>
  <state id = “S2” initial = “S21” >
        ......................
        ......................
  </state>
</parallel>
```

Two parallel regions start with sub states S11 and S21. The internal transitions are given between <state? And </state> tags.

During code generation the parsed XML document is given to the code generator. The code generator has mainly three modules. One module analyzes the parsed XML and find out the state nodes. It generates one state class for each state node based on the design pattern in the pattern library.

The composite states are implemented as the extension of State abstract class and the OrthogonalProperty class. The nested states are implemented as the derived classes of the corresponding composite states.

For the XML document given in the tree view (Figure 18) contains two states; off and on. So, two state classes will be generated; offState and onState. The on state contains some sub states. It shows that onState is a composite state. Since <parallel> tag is not there, it is understood that the sub states are not concurrent states. So, the on state will be defined as OnCompoState class by extending the State class and the



```
⊟ state ..
    @id: off
  ⊞ transition
⊟ state ..
    @id: on
  ⊟ initial ..
    ⊞ transition
  ⊞ transition
  ⊞ transition
  ⊟ state ..
      @id: idle
    ⊞ transition
    ⊞ transition ..
  ⊟ state ..
      @id: cooking
    ⊞ transition ..
    ⊞ transition ..
```

FIGURE 18. XML representation of Microwave Oven.

OrthogonalProperty class. offState will be defined as offState class. The inner states idle and cooking will be defined as derived classes of the OnCompoState class; idleState class and cookingState class.

Similarly, the parallel states can be identified by the tag <parallel> as in figure 19. Parallel states are composite states by default. The number of states inside the <parallel> tag will help us to set the number of regions. In the example we have two states inside the <parallel>, that means two parallel regions.

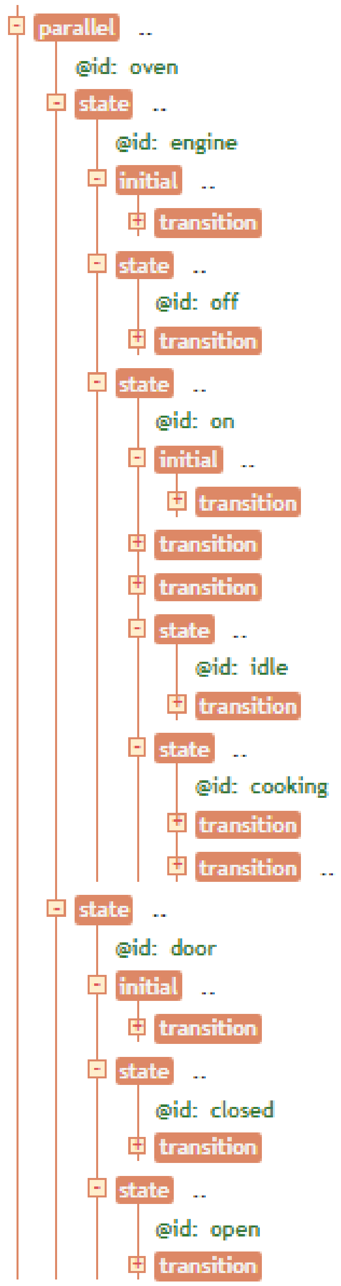
The second module in the code generator analyzes the events and transitions. It generates the event class and updates the event dispatch functions in each state class.

The event is stored as the attribute of the element <transition>. For example, <transition event = "e" target = "s2"/>. So all <transition> elements have to be checked and list out all those events in the Event class by eliminating the duplications.

```
<state id = "S2" initial = "S21" >
        <state id = S21">
          <transition event ="e1" target = "S22"/>
        </state>
        <state id = "S22" >
          <transition event = "e2" target = "S2Final/>
        </state>
        <final id = "S2Final"/>
    </state>
```

In the above XML statements, a composite state with two sub states is given. Two transitions have been described in this composite state. In each <transition> we can identify an event; say e1 and e2. These events will be added to the Events class. If already existing event (in the Event class) is found, it will be ignored.

The event dispatch function will be updated with the transition and the target state. For example, in state S21, there is a transition to state S22 when an event e1 occurs. So, the dispatch function of the class S21State will be updated by adding the corresponding 'case' statement and the state transition with a call to transition() function. The activity inside each state will be converted as the program statements and added to the dispatch function of the corresponding state class.



```
⊟ parallel ..
    @id: oven
  ⊟ state ..
      @id: engine
    ⊟ initial ..
      ⊞ transition
    ⊟ state ..
        @id: off
      ⊞ transition
    ⊟ state ..
        @id: on
      ⊟ initial ..
        ⊞ transition
      ⊞ transition
      ⊞ transition
      ⊟ state ..
          @id: idle
        ⊞ transition
      ⊟ state ..
          @id: cooking
        ⊞ transition
        ⊞ transition ..
  ⊟ state ..
      @id: door
    ⊟ initial ..
      ⊞ transition
    ⊟ state ..
        @id: closed
      ⊞ transition
    ⊟ state ..
        @id: open
      ⊞ transition
```

FIGURE 19. XML representation of Microwave Oven with concurrent states.

The third module in the code generator analyzes the state transitions and its flow. It generates the context class for the system. The context class represents the entire system, or the system state chart. It receives the events and delegates the actions to the corresponding active state. The action to be done on a particular transition is defined in the dispatch function of each state class.

VIII. RELATED WORKS

The code generation from UML diagrams is an ever growing research area. Many authors contributed to this [1], [11], [12], [14], [16], [23] even from late 90s.

Pais and Oliveira [52] proposes some stereo types for UML to specialize the boundary, control and entity elements or classes. The robustness diagram represented using these stereo types can be converted to UML state chart diagrams, sequence diagram and class diagram. Kundu *et al.* [54] proposes code generation method from UML sequence diagrams. The sequence diagrams first converted to sequence interactions graphs. These graphs contain information like messages, control flow and method scope of interactions. These graphs will then be transformed to code. This method is best suited for the controller classes and not for boundary and entity classes. Sunitha and Samuel [53] presents the formal association between activity diagram and sequence diagram. Based on this formal definition, the paper proposes a method to generate code from UML activity diagram and sequence diagrams. This method is best suited for boundary and control classes.

But, in this paper we proposed code generation from UML state chart diagrams. Since the state chart diagram shows the state of the entity classes (objects), the method proposed in this paper is best suited for entity classes and not for boundary and controller classes.

Pham *et al.* [55] presented a code generation method from the UML state chart diagrams based on "if..else" statements.

Niaz *et al.* [3], Niaz [15] propose a method to implement composite and concurrent states. In the proposed approach single event can trigger multiple transitions. This is against the semantics of the UML state machine. UML specifies that one event should be consumed for only one transition.

Harrand *et al.* [48] proposes ThingML to utilize the benefits of the Model based system engineering (MBSE). ThingML is a domain specific modeling language (DSML). It supports automated code generation from system models, thereby increases the productivity of the software development team. Harrand *et al.* [48] states that they implement the behavior of a system using code generation from the state machines of the system. For code generation, they use existing design patterns in C++ or Java, or else other existing frameworks such as State.js. They focus on heterogeneous target platforms.

Schattkowsky and Muller [17] demonstrates how a fully featured UML 2.0 state machine can be represented using a small subset of the UML state machine features that enables efficient execution. They are trying to directly execute the state machines without converting it to implementation code. It is an alternative to native code generation approaches since it significantly increases portability. The paper describes the necessary model transformations in terms of graph transformations and discusses the underlying semantics and implications for execution.

Rudhal and Goldin [18] presents a multi language code generator named as YAMDAT (Yet Another MDA Tool). As the name indicates, it's an MDA tool. It generates C++ and Java code from UML designs of the system. UML models will be represented in XML and this XML representation is the code model in the tool. They generate skeleton code for all methods and attributes in the UML class diagram. Moreover, unit test framework will be generated for the class. YAMDAT generates finite state machine class from each state diagram of the class.

Dominguez *et al.* [26] presented a review of research works that propose methods to implement UML state chart diagrams. Dominguez *et al.* summarizes their review by saying that the state transition process in most of the works are based on switch statement or state table. Another key finding of [26] is that very few papers support hierarchy and concurrency of states.

Ali *et al.* [6] implements state as objects and the state hierarchy and concurrency are implemented using inheritance and composition. They introduced a concept of helper object to handle the state specific requests. it works as the context class in the state design pattern based methods.

Ali and Tanaka [9] proposed a method to implement the dynamic behavior of an application. State transition diagrams and activity diagrams are used for modeling the dynamic behavior. The state of the system is represented as object and the state transition is implemented as method. The state hierarchy and concurrency are implemented using inheritance and composition.

Ali and Tanaka [20] presents a method to convert the dynamic model to executable code. State of the system is represented as object, and the events are the methods in the state class. They developed a tool called O-Code based on their proposed method. The input to the tool is state diagram represented in Design Schema List (DSL). O-Code has two parts, one is an interpreter and the other one is the code generator. The interpreter generates a transition table from the DSL while code generator generates Java code from the transition table.

Ali [27] presents the implementation of concurrent and hierarchical state machines by making use of enumerators in Java language. It proposes a Java implementation pattern for state machines. The method presented in the paper is language dependent and so it cannot be extended to other languages like C++, C# etc.

Aabidi *et al.* [25] proposed a method to implement hierarchical-concurrent states using Java Enum. It combines the methods proposed in [2] and [27] to provide a better way to implement state machines leveraging the positive points of both approaches, state machine encapsulated within a single class, code well structured, clear, compact and easy to understand for the first approach and a better identification of the state for the second approach.

Spinke [24] addresses concurrent and hierarchical state implementation. The paper proposed a double dispatch based event handling. The reaction of the state machine depends on the current state of the system as well as the event occurred. This is the theory behind double dispatch. The paper presents a case study to show case the proposed method. The implementation pattern presented in the paper is very bulky since it requires 17 classes in the implementation for representing a state machine with 6 states and 6 events.

Samek [29] hierarchical state machine pattern (HSM) to implement state hierarchy and transition dynamics. They presented the methods to convert the state machine to C or C++ code. The statemachines are defined as a composition of states. HSM pattern uses state classes for composition not for inheritance or sub classing. They also presented Quantum Hierarchical State Machine pattern (QHsm) [30]. It used Quantum Programming paradigm. They presented the modeling of state charts in C/C++ using HSM and active-object based framework.

Heinzmann [31] extends the HSM pattern method. It presents a template based approach to directly convert the state chart diagrams to the C++ code. This method avoids the use of separate code generation tools for the state machine to code conversion. The generated code is optimized using inlining.

Breti [4] presented a method to generate C++ code from the State chart. State chart is modeled in XML and then using Python the XML document is parsed to Python object. in the next step a templating engine is used to convert the parsed XML to C++ code. For this conversion they use a pattern based approach. The pattern contains state controller, state chart, state, and events. The implementation details of the state hierarchy and concurrency are not specified in this paper.

The Object Management Group (OMG) has specified a subset of UML 2.0 exclusively for Model Driven Development to support executable UML (xUML). This subset is named as Foundational UML (fUML) [49]. fUML considers only class diagram and activity diagram for the xUML. It does not support state diagram and sequence diagram. fUML specifies semantics for xUML. In order to improve the precision an action language named Alf [56] is used with fUML. It is a textual action language. However, it does not have any advantage over UML and OCL, since fUML requires detailed modeling and precisions should be added using an action language like OCL. A sound knowledge in fUML and Alf is necessary to convert fUML models to code. The same thing can be done with UML and OCL and with less effort since most of the developers and designers are familiar with those standards.

IX. EVALUATION AND COMPARISON WITH RELATED WORKS

The proposed pattern is implemented using the code generator called SMConverter. The performance of SMConverter is compared with other tools like Rhapsody [28] and OCode [9], [20]. We considered the events with and without transitions. These events have been passed to the AlarmClock class in the form of sequence of requests. Total time taken for each type of event is calculated in milliseconds. Total number of requests for events without transition is 1778 and for events with transition is 2222. The efficiency of our tool (SMConverter) over other tools is shown in the table 3. Figure 20 compares the total time taken for events without and with transition respectively.

TABLE 3. Efficiency of SMConverter compared with Rhapsody & OCode.

	Ocode (ms)	Rhapsody (ms)	SMConverter (ms)	Effieciency over Ocode	Efficiency over Rhapsody
Total time for events without transitions	8.8	4.3	3.55		
Average time per event without transition	0.004949	0.002418	0.001997	59.66	17.44
total time for events having transitions	28.3	19.35	11.35		
average time per event having transition	0.012736	0.008708	0.005108	59.89	41.34
total time for all events	37.1	23.65	14.9		
average time per event	0.009275	0.005913	0.003725	59.84	37

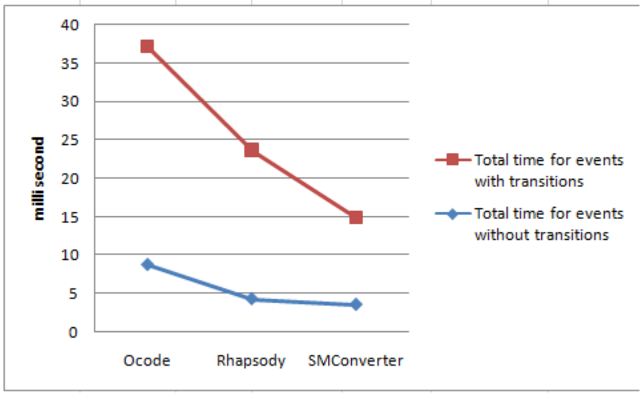


FIGURE 20. Total time for events without transition.

The proposed approach is compared with 10 major research works in this area [3], [5], [6], [9], [15], [20], [24], [29]–[31]. For the comparison, we considered the method of implementation of different elements in the state chart diagrams like simple state, composite state, history state etc., and the support for different features of state machines like, hierarchy, concurrency etc. The details of comparison are given in tables 4 and 5.

A. ELEMENT BASED COMPARISON

For element based comparison, we considered the basic elements like, current state, state transition etc. in state machines and the components like, orthogonal states, composite states,

TABLE 4. Element based comparison with related works.

Elements\ Reference	Tanaka [3,6,9,15,20]	Douglass[5]	Spinke[24]	Samek[29]	Samek[30]	Heinzmann[31]	Proposed method
Context Class	class	class	subclass	sub class	sub class	class	sub class
Current State	attribute	attribute	attribute	attribute	attribute	attribute	attribute
Simple State	separate class for each state	defining a single state class for all states	separate class for each state	defining a single state class for all states	member functions for each state	defining a single state class for all states	separate class for each state
state transition process	switch stmt	state-table structure	double-dispatch	switch stmt	switch stmt	switch stmt	switch stmt
simple Composite state	derived class of abstract state class	X	derived class of abstract state class	a class state for all states	member functions	a template class	derived class of Orthogonal State class (no of regions = 1)
Orthogonal Composite state	derived class of abstract state class	X	A class for all orthogonal states, derived from abstract state class	X	X	X	derived class of Orthogonal State class
Shallow History	attribute	X	attribute	attribute	X	X	attribute
Deep History	X	X	attribute	X	attribute	X	History State Class

TABLE 5. Feature based comparison with related works.

Features\ Reference	Tanaka [3,6,9,15,20]	Douglass[5]	Spinke[24]	Samek[29]	Samek[30]	Heinzmann[31]	Proposed Method
hierarchy	supported	not supported	*supported	supported	supported	supported	supported
concurrency	supported	not supported	supported	not supported	not supported	not supported	supported
history	partially supported	not supported	supported	partially supported	partially supported	not supported	supported
expandability	good	bad	bad(no pattern)	good	medium	medium	good
Simplicity	yes		no	neutral	yes	neutral	yes
reusability	medium		bad(no pattern)				good
understandability	good	bad	medium	medium	good	medium	good
modularity	good	good	good	good	good	medium	good

history states etc., that improve the expressive richness of the state machine. We studied how each of these elements is implemented in the present literatures and how well they support object orientation. The current state, simple states and the context class are represented in a similar way in all the literatures. In case of state transition all literatures except [24] uses switch statement. The components like, composite, orthogonal, and history states are supported by few research works.

The element based comparison, given in table 4, shows that our method implements the state chart elements in object oriented way, but many of the related works do not. In our method, we implement the concurrent states, composite states and deep history states as classes and their attributes. So our method can take the advantages of any object oriented approach.

B. FEATURE BASED COMPARISON

In feature based comparison, we considered the features like, expandability, reusability, understandability etc., and the support for state hierarchy and concurrency in the state chart diagrams.

The state hierarchy is supported by almost all works, except [5], but concurrency is not well supported by the research works. History of state machine state is also not supported by many literatures. Many research outcomes provide methods with good modularity and understandability, but they failed to provide reusability due to the lack of general reusable design pattern. Our proposed method gives a design pattern and which gives good understandability, reusability and expandability.

Feature based comparison, given in table 5, shows that our method supports hierarchy and concurrency without spoiling the understandability, reusability and expandability.

X. CONCLUSION

In this work, we introduced a design pattern based implementation of state machine with hierarchical, concurrent and history states. The design pattern proposed in the paper provides modularity and understandability. Moreover it keeps the semantics of state hierarchy and concurrency and history state as well.

The design pattern is easily expandable due to its modular structure. The use of design pattern in code generation improves the quality of the generated code. Most of the research works do not provide implementation of concurrent and history states. Many research works do not provide a general design pattern for the state machine implementation. Meanwhile, we provide a reusable and understandable design pattern.

The code generator presented in the paper gives a systematic way of code generation from the UML state chart diagrams with the help of the proposed design pattern.

Comparison with related works shows that the proposed way of state chart diagram implementation supports the important features like concurrency and history. Moreover, the qualitative comparison with the related works shows that our method supports state concurrency and history without compromising the expandability, reusability and understandability.

Comparison with other tools shows that our method is more efficient in terms of the time taken for event processing. The case study and comparison with other tools reveals that the proposed approach gives less complex code and promising results.

REFERENCES

[1] L. Lazareviae and D. Miliaev, "Finite state machine automatic code generation," in *Proc. IASTED Conf.*, Las Vegas, NV, USA, 2000, pp. 1–3.

[2] A. Jakimi and M. Elkoutbi, "Automatic code generation from UML state chart," *Int. J. Eng. Technol.*, vol. 1, no. 2, pp. 165–168, Jun. 2009.

[3] I. A. Niaz and J. Tanaka, "An object-oriented approach to generate Java code from UML statecharts," *Int. J. Comput., Inf. Sci.*, vol. 6, no. 2, pp. 315–321, 2005.

[4] J. Breti, *State Machine Code Generation in Python*, document version 1.0.1, Gnosis Version, Canada, 2007.

[5] B. P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, MA, USA: Addison-Wesley, 1998.

[6] J. Ali and J. Tanaka, "Converting statecharts into Java code," in *Proc. 5th Int. Conf. Integr. Design Process Technol. (IDPT)*, 1999, p. 42.

[7] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Reading, MA, USA: Addison-Wesley, 1999.

[8] J. Rumbaugh, I. Jacobson, and G. Booch, *Object-Oriented Analysis and Design With Applications*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2007.

[9] J. Ali and J. Tanaka, "Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams," *J. Comput. Sci., Inf. Manage.*, vol. 2, no. 1, pp. 24–34, 2001.

[10] M. J. Balcer and S. J. Mellor, *Executable UML A Foundation for Model-Driven Architecture*. Reading, MA, USA: Addison-Wesley, 2002.

[11] S. Sengupta, A. Kanjilal, and S. Bhattacharya, "Automated translation of behavioral models using OCL and XML," in *Proc. IEEE Region TENCON*, Nov. 2005, pp. 1–6.

[12] X. Li and Z. Liu, "Prototyping system requirements model," *J. Electron. Notes Theor. Comput. Sci.*, vol. 207, pp. 17–32, Apr. 2008.

[13] M. Usman, A. Nadeem, and T.-H. Kim, "UJECTOR: A tool for executable code generation from UML models," in *Proc. Adv. Softw. Eng., Appl.*, 2008, pp. 165–170.

[14] U. A. Nickel, J. Niere, R. P. Wadsack, and A. Zündorf, "Roundtrip engineering with FUJABA," in *Proc. 2th Workshop Softw.-Eng.*, Bad Honnef, Germany, 2000, pp. 36–39.

[15] I. A. Niaz, "Automatic code generation from UML class and statechart diagrams," M.S. thesis, School Syst. Inf. Eng., Univ. Tsukuba, Tsukuba, Japan, 2005.

[16] W. Harrison, C. Barton, and M. Raghavachari, "Mapping UML designs to Java," in *Proc. 15th ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2000, pp. 178–187.

[17] T. Schattkowsky and W. Muller, "Transformation of UML state machines for direct execution," in *Proc. IEEE Symp. Vis. Lang. Hum.-Centric Comput. (VLHCC)*, Sep. 2005, pp. 117–124.

[18] K. T. Rudhal and S. E. Goldin, "Adaptive multi-language code generation using YAMDAT," in *Proc. 5th Int. Conf. ECTI-CON Elect. Eng./Electron., Comput., Telecommun. Inf. Technol.*, vol. 1, May 2008, pp. 181–184.

[19] P. Samuel and E. V. Sunitha, "Automatic code generation using model driven architecture," in *Proc. IEEE Int. Adv. Comput. Conf. (IACC)*, Patiala, India, Mar. 2009, pp. 2339–2344.

[20] J. Ali and J. Tanaka, "An object oriented approach to generate executable code from OMT-based dynamic model," *J. Integr. Design Process Sci.*, vol. 2, no. 4, pp. 65–77, 1998.

[21] E. V. Sunitha and P. Samuel, "Translation of behavioral models to source code," in *Proc. 12th Int. Conf. Intell. Syst. Design Appl. (ISDA)*, Kochi, India, 2012, pp. 598–603.

[22] OMG. (Jun. 2003). *MDA Guide Version 1.0.1*. [Online]. Available: http://www.omg.org

[23] Q. Long, Z. Liu, X. Li, and H. Jifeng, "Consistent code generation from UML models," in *Proc. Austral. Softw. Eng. Conf.*, 2005, pp. 23–30.

[24] V. Spinke, "An object-oriented implementation of concurrent and hierarchical state machines," *J. Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1726–1740, 2013.

[25] M. H. Aabidi, A. Jakimi, R. Alaoui, and E. H. El Kinani, "An object-oriented approach to generate Java code from hierarchical-concurrent and history states," *Int. J. Inf. Netw. Secur.*, vol. 2, no. 6, pp. 429–440, 2013.

[26] E. Domínguez, B. Pérez, Á. L. Rubio, and M. A. Zapata, "A Systematic review of code generation proposals from state machine specifications," *J. Inf. Softw. Technol.*, vol. 54, no. 10, pp. 1045–1066 2012.

[27] J. Ali, "Using Java Enums to implement concurrent-hierarchical state machines," *J. Softw. Eng.*, vol. 4, no. 3, pp. 215–230, 2010.

[28] I-Logix Inc. *Rhapsody*. Accessed: Mar. 2010. [Online]. Available: http://www.ilogix.com

[29] M. Samek, *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. Newton, MA, USA: Newnes, 2008.

[30] M. Samek, *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems With CDROM*. Lawrence, KS, USA: CMP Books, 2002.

[31] S. Heinzmann, "Yet another hierarchical state machine," *Overload J., Assoc. C&C++ Users*, no. 64, pp. 14–21, 2004.

[32] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Delhi, India: Pearson Education, 1995.

[33] J. Wang, Ed. *Handbook of Finite State Based Models and Applications*. Boca Raton, FL, USA: CRC Press, 2012.

[34] M. L. Alvarez, I. Sarachaga, A. Burgos, E. Estévez, and M. Marcos, "A methodological approach to model-driven design and development of automation systems," *IEEE Trans. Autom. Sci. Eng.*, vol. 15, no. 1, pp. 67–79, Jan. 2016.

[35] K. Lano and S. Kolahdouz-Rahimi, ''Model-transformation design patterns,'' *IEEE Trans. Softw. Eng.*, vol. 40, no. 12, pp. 1224–1259, Dec. 2014.
[36] D. Milicev, ''Automatic model transformations using extended UML object diagrams in modeling environments,'' *IEEE Trans. Softw. Eng.*, vol. 28, no. 4, pp. 413–431, Apr. 2002.
[37] S. Sendall and W. Kozaczynski, ''Model transformation the heart and soul of model-driven software development,'' *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, 2003.
[38] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, ''Model-driven development using UML 2.0: Promises and pitfalls,'' *Computer*, vol. 39, no. 2, pp. 59–66, Feb. 2006.
[39] E. V. Sunitha and P. Samuel, ''Object oriented method to implement the hierarchical and concurrent states in UML state chart diagrams,'' in *Software Engineering Research, Management and Applications*. Springer, 2016, pp. 133–149.
[40] T. C. Lethbridge and R. Laganiere, *Object-Oriented Software Engineering*. New York, NY, USA: McGraw-Hill, 2005.
[41] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Upper Saddle River, NJ, USA: Prentice-Hall, 2004.
[42] P. Wolfgang, *Design Patterns for Object-Oriented Software Development*. Reading, MA, USA: Addison-Wesley, 1994.
[43] D. Harel and O. Kupferman, ''On object systems and behavioral inheritance,'' *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, pp. 889–903, Sep. 2002.
[44] D. Harel and A. Naamad, ''The Statemate semantics of statecharts,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 293–333, 1996.
[45] R. Bendraou, J.-M. Jezequel, M.-P. Gervais, and X. Blanc, ''A comparison of six UML-based languages for software process modeling,'' *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 662–675, Sep./Oct. 2010.
[46] F. R. Wagner and L. Carro, ''Embedded SW design space exploration and automation using UML-based tools,'' in *Embedded System Design: Topics, Techniques and Trends*. New York, NY, USA: Springer, 2007, pp. 437–440.
[47] E. Gamma, *Design Patterns: Elements of Reusable Object-Oriented Software*. Delhi, India: Pearson Education, 1995.
[48] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, ''ThingML: A language and code generation framework for heterogeneous targets,'' in *Proc. ACM/IEEE 19th Int. Conf. Model Driven Eng. Lang. Syst.*, Oct. 2016, pp. 125–135.
[49] Object Management Group Standard. (2013). *Semantics of a Foundational Subset for Executable UML Models, Version 1.2.1*. [Online]. Available: http://www.omg.org/spec/FUML/
[50] J. Barnett, *State Chart XML (SCXML) State Machine Notation for Control Abstraction. W3C Recommendation*, document REC-scxml-20150901/, 2015.
[51] E. Sekerinski, ''Design verification with state invariants,'' in *UML 2 Semantics and Applications*. Hoboken, NJ, USA: Wiley, 2009, pp. 317–347.
[52] A. P. P. Pais and C. E. Oliveira, ''Enhancing UML expressivity towards automatic code generation,'' in *OOIS*. London, U.K.: Springer, 2001, pp. 335–344.
[53] E. V. Sunitha and P. Samuel, ''Automatic code generation using unified modeling language activity and sequence models,'' *IET Softw.*, vol. 10, no. 6, pp. 164–172, 2016.
[54] D. Kundu, D. Samanta, and R. Mall, ''Automatic code generation from unified modelling language sequence diagrams,'' *IET Softw.*, vol. 7, no. 1, pp. 12–28, Feb. 2013.
[55] V. C. Pham, A. Radermacher, S. Gérard, and S. Li, ''Complete code generation from UML state machine,'' in *Proc. 5th Int. Conf. Model-Driven Eng. Softw. Develop. (MODELSWARD)*, Porto, Portugal, Feb. 2017, pp. 208–219.

SUNITHA E. V. received the B.Tech. degree in IT and the M.Tech. degree in software engineering from the Cochin University of Science and Technology, Kochi, where she is currently pursuing the Ph.D. degree. She is also an Assistant Professor with the IT Department, Toc H Institute of Science and Technology, Kerala. Her research interests include software engineering and object oriented modeling.

PHILIP SAMUEL received the M.Tech. degree in computer science from the Cochin University of Science and Technology (CUSAT), Kochi, and the Ph.D. degree from IIT Kharagpur, India. He is currently a Professor with the Department of Computer Science, CUSAT. His research interests include software engineering, object oriented modeling and design, mobile communication, the IoT, and adhoc networks. He had published several research papers in these areas.