CODE GENERATION FROM UML STATECHARTS

Iftikhar Azim Niaz and Jiro Tanaka
Institute of Information Sciences and Electronics
University of Tsukuba
Tennoudai 1-1-1, Tsukuba, Ibaraki 305-8573 Japan
{ianiaz, jiro}@iplab.is.tsukuba.ac.jp

ABSTRACT

The Unified Modeling Language (UML) statechart diagram is a powerful tool for specifying the dynamic behavior of reactive objects. Generating code from statechart diagrams is a challenging task due to its dynamic nature and because many of the statechart concepts are not supported by the object oriented programming languages. Most of the approaches for implementing UML statecharts diagram either suffer from maintenance problems or implement only a subset of UML statecharts.

This paper proposes a new approach to generate efficient and compact executable code from the UML statechart diagram in an object-oriented language like Java using design patterns. In our approach, each state in the statechart becomes a class, which encapsulates all the transitions and actions of the state. The events that have transitions are thus made explicit without using any *if* or *case* statement, which leads to a readable, compact and efficient code. The resultant code is easy to maintain.

By representing states as objects, we extend the state design pattern to implement the sequential substates and concurrent substates using the concept of object composition and delegation. We also propose an approach to implement compound transitions (fork/join) and history nodes. The proposed approach makes elegant handling of most of the statechart features.

KEY WORDS

Software engineering, Object-oriented analysis and design, Statecharts, State pattern, Object composition, Java

1. Introduction

The emergence of the UML [1] as an industry standard for modeling systems has encouraged the use of automated software tools that facilitate the development process from analysis through coding. In UML based object-oriented design, behavioral modeling aims at describing the behavior of objects using state machines. A state machine is a behavior that specifies the sequences of states an object goes through its lifetime in response to events, together with its responses to those events. The UML statechart diagram is a graph that represents a state machine [1]. The semantics and notation used in UML statecharts mainly follow Harel's statecharts [2] with extensions to make them object-oriented. A statechart attached to a class specifies all behavioral aspects of the objects in that class.

The OO methodologies using statecharts describe in sufficient detail the steps to be followed for describing the behavior of objects but fail to describe the implementation of statecharts in the object-oriented languages due to lack of syntactic support for statecharts. There exists a gap between high-level modeling language and a programming language. The translation of class diagrams to an OO programming language is easy and provided by most CASE tools. To implement the behavior of an object-oriented system, one has to implement the statecharts, which specify the dynamic behavior of the classes. Our approach is an effort to bridge the gap between design and implementation. Through mapping between UML and Java, we are able to generate low-level java code directly from the statechart diagram. The primary goal is to present a simple and efficient implementation of the UML statecharts in an object oriented language like Java. The proposed implementation techniques are valuable in that they raise the level of abstraction and allow for straightforward mapping of statecharts to compact and efficient code.

There are number of ways to implement a statechart in OO programming languages. The most common technique to implement statechart is the doubly nested *switch* statements with a "scalar variable" used as the discriminator in the first level of the switch and event-type in the second [3]. This works well for simple statecharts but manual coding of entry/exit actions and nested substates is cumbersome, mainly because code pertaining to one state becomes distributed and repeated in many places, making it difficult to modify and maintain when the topology of the statechart is changed. In [4] and [5] the relation between states and classes was examined. They proposed to reuse behavior in state machines through inheritance of other state machines. They implemented embedded states by making a table for the superstate and did not consider concurrent states, history states and compound transitions. In [6] all states from the statechart were generated as constant attributes in the class and another attribute was used to keep track of the current state of an object. Transition of states was

represented by the *state* attribute of an object changed into another new state's value. Events and actions were implemented as methods. Action was translated into method call in the code. It only handled the simple statechart diagram.

Design pattern approach [7] is widely used in object-oriented software design. The approach specifies reusable mechanisms for collaboration and interaction among classes or among objects to solve common object-oriented problems in any domain. Several design patterns have been proposed to implement statecharts. *Conditional Statements* pattern [3], *StateTable* pattern [3], *Basic Statechart* pattern [8], HSM pattern [9] and *State* pattern [7]. These patterns have critical problems with support of substates, mapping from diagram to code and the readability of code. HSM pattern only supports the sequential substates.

Our work focuses on the implementation phase. We are looking for a code generator tool to allow automatic translation of statechart diagrams to object-oriented programming languages like java. This paper proposes a new approach to narrow the gap between UML statechart diagram and an implemented system. The narrowing of gap is achieved by generating low-level Java code from UML statecharts. The code generation is achieved by creating a mapping between UML statecharts and Java programming language.

In this paper, we will describe our approach for implementing the sequential substates, concurrent substates and compound transitions in Java by extending State pattern with object composition and delegation.

2 Sequential Substates

We will use an air conditioner system as an example to demonstrate our approach.

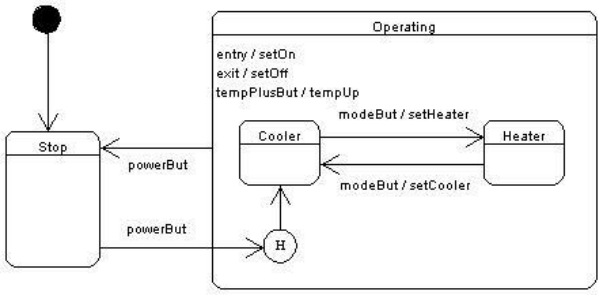

Figure 1. Statechart for air conditioner system

The basic behavior of an air conditioner is specified in the statechart shown in Figure 1. It has two top-level states *Stop* and *Operating*. Initially, air conditioner is in state *Stop*, where it accepts the powerBut event. The air conditioner reacts on such an event by switching from the *Stop* state to *Operating* state. The *Operating* state is the composite state with *Cooler* as its initial substate. *Heater* state is the other sequential substate. While in *Operating* state, on modeBut event, the air conditioner switches to the next substate. On powerBut event, the air conditioner goes to the *Stop* state. Sending a powerBut event may reactivate the air conditioner. When the air conditioner is reactivated, it switches into the history state of the *Operating* state. The history state stores the last substate that was active before it switches from *Operating* to *Stop* state and recalls the substate when the air conditioner is reactivated. Transitions affecting the superstate apply at all levels of nesting within that superstate.

2.1 Implementation Strategy

Our approach for implementing UML statechart diagram is based on [10], [11], [12] and the *State* pattern [7]. State pattern puts all behavior associated with a particular state into one class. In our approach, each state in the statechart diagram becomes a class. Each transition from that state becomes a method in the corresponding class and each action becomes a method in the context class whose behavior is represented by the statechart diagram. A context class delegates all events for processing to the current state object. State transitions are accomplished by changing the current state object. The *State* pattern does not deal with the sequential substates and concurrent substates. So we need some mechanism to implement the state hierarchy. Object composition is defined dynamically at runtime through objects acquiring references to other objects. Object composition keeps each class encapsulated and there are substantially fewer dependencies. The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they are composed.

When a composite state is active, exactly one of its sequential substates is active. The sequential substates show a nested statechart within the composite state. This leads us to implement the composite state by extending the state design pattern with object composition and delegation. In this case the composite state becomes the context for the nested statechart. An abstract state class will define the interface for the behavior associated with sequential substates of the composite state. The sequential substates will become the concrete state classes. The composite class will keep the control most of the time and delegates the requests to substates for transitions specific to substates. Figure 2 shows the class diagram of our implementation approach.

The AirCon class is the super context class with which the statechart of Figure 1 is associated. Each action becomes a method of the context class. The state object will hold the reference of the current active state. The history node is implemented by providing a reference opHistory in the AirCon object, which sets the opHistory at the start to *Cooler* state and later it is adjusted to the current active substate in the exit() method by the composite state *Operating*. When *Operating* state

becomes active, its most recent active substate is also set. The AirConState and AbsOpState classes are the abstract state classes, which provide a common interface for the concrete state classes. All the concrete state classes have a reference to the context object. The AirCon (context) object delegates all incoming events to its current state object (state). On handling the transition, the concrete state object first executes the exit action of the current state and then calls the setState() method of the AirCon object to set the new state. In the setState() method, the entry action of the new state is also executed. Operating class becomes context for the nested statechart. The subState object in the Operating object will keep the reference of the current active substate. Cooler and Heater objects will maintain two references for the two contexts Operating and AirCon. If the target of a transition is the composite state then it is executed by the composite state but if the target is the substate then the composite state object will delegate the request to the active substate. The active substate will execute the corresponding action on the transition and then executes the exit action and then sets the next substate by calling the setSub() method of the composite Operating object. The internal transition *tempPlusbut* is implemented in the corresponding Operating class. The action tempUp becomes a method in the context class AirCon.

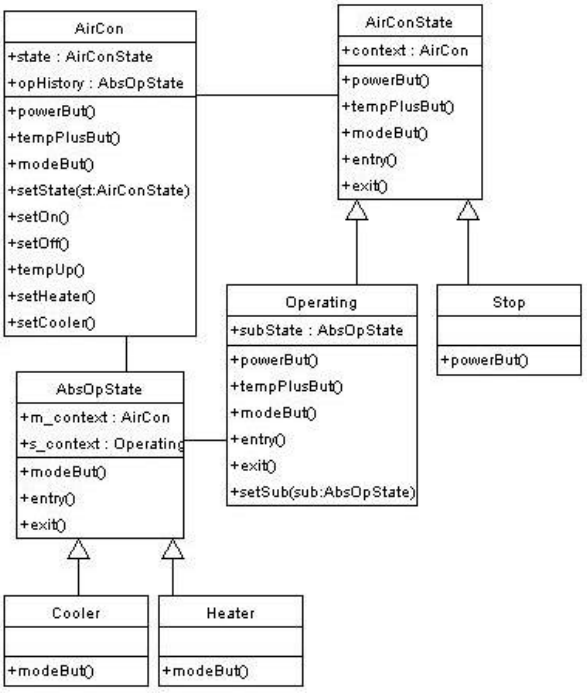

Figure 2. Class diagram for implementing sequential substates

3 Concurrent Substates

In this section, we describe our approach for implementing concurrent substates. When a composite state contains concurrent (orthogonal or AND) substates, the substates become active simultaneously whenever the composite state become active. These substates specify two or more state machines that executes in parallel in the context of enclosing object. Figure 3 shows a more detailed statechart containing concurrent substates for the Air Conditioner. *Mode* and *Speed* are the two concurrent regions of *Operating* composite state.

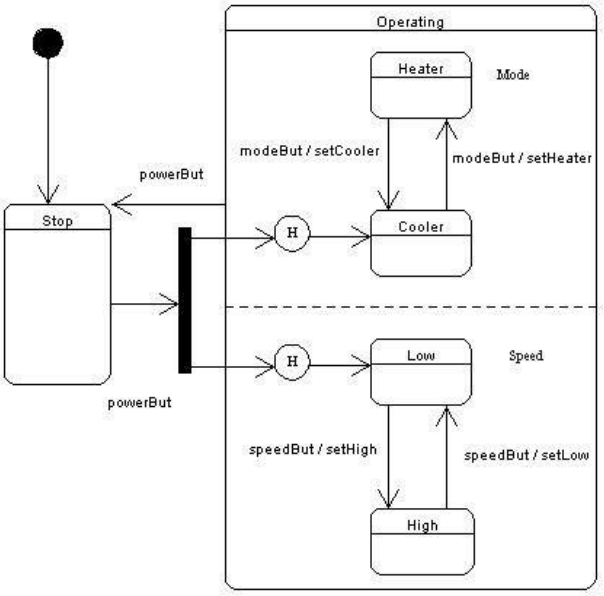

Figure 3. Statechart having concurrent states

3.1 Implementation Strategy

The concurrent substates show a nested statechart within composite state. This leads us to implement the concurrent substates by extending the state pattern with object composition and delegation. The *Operating* state will become the context for both concurrent regions *Mode* and *Speed*. *Mode* and *Speed* will become the abstract classes and will define the interface for the behavior associated with nested sequential substates within each concurrent region. Figure 4 shows the class diagram of our implementation approach.

As can be seen in Figure 4, Operating class becomes the context for the two concurrent regions *Mode* and *Speed*. AbsModeState and AbsSpeedState classes provide the interface for the two concurrent regions. Cooler, Heater, Low, and High become the concrete substate classes for Operating composite state. The Operating object will keep the references of the current active substate within each concurrent region in modeState and speedState objects. The concrete state objects will maintain two references for the two contexts Operating

and AirCon. The history nodes are implemented by providing references modeHistory and speedHistory in the AirCon class. The *powerBut* event from the *Stop* state forks into two concurrent regions of *Operating* state so the *Stop* state is responsible for activating the proper substates of the two concurrent regions. It calls the fork method of the context, which makes all the target states active rather the one. On receiving the *modeBut* or *speedBut* request in the *Operating* state, the Operating object will delegate the request to the current active substate. The active substate will execute the corresponding action on the transition and then change the substate by calling the appropriate set substate method of the Operating object.

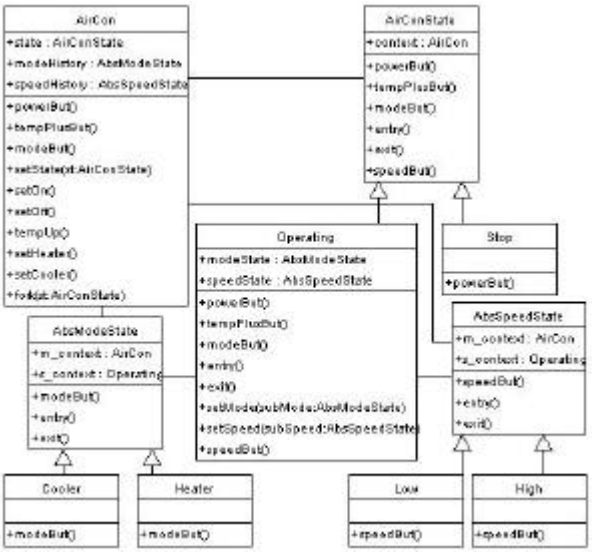

Figure 4. Class diagram for implementing concurrent substates.

4 Compound Transitions

A compound transition represents a path made of one or more transitions, originating from a set of states and targeting a set of states. A compound transition is enabled when all the source states are occupied. After a compound transition fires, all of its destination states are occupied. A compound transition is shown as a short bar.

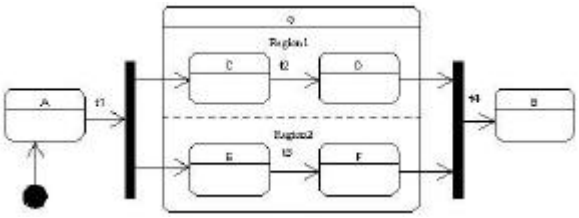

Figure 5. Statechart having compound transitions

Fork

A fork is a transition with one source state and two or more target states. If the source state is active and the trigger event occurs, the transition action is executed and all the target states become active.

Join

A join is a transition with two or more source states and one target state. If all the source states are active and the trigger event occurs, the transition action is executed and the target state becomes active.

Figure 5 shows a statechart having compound transitions. The trigger event for the fork is t1. This means when the A state is active and the event t1 occurs, both C and E states will become active simultaneously in the two concurrent regions. Figure 5 also contains a join going from D and F states to the B state. Transition t4 will be fired only if D and F are active. If D and F states are not active then the event is ignored.

4.1 Implementation Strategy

Implementing fork is easy. The source state makes all the target states active rather than the one. To implement join we have to make sure that all the source states must be active before the transition fires. The Java implementation code for Figure 5 can be written as follows. We suppose that the context class name is Test.

```
class Test {
  TestState  state;    // state object
    // References for all the state objects
  A  aState; B  bstate; G  gState; C cState;
  D  dState; E  eState; F  fState;
  void setState(TestState st) {// setting new state
            state = st;
            state.entry(); } // executes the entry action
  void fork(TestState st) { // setting the concurrent states
    if (st.equals(gstate)) {
     gstate.region1State = cstate;
     gstate.region2State = estate; }
…….
}  // End of Test Class
class TestState {
            Test  ac;  // Reference to the context object
    // declaring abstract methods
}
class A extends TestState {
  void t1() { // implementing Fork
    exit();  // executes the exit action of current state
    ac.fork(ac.gState);// setting substates to be active
    ac.setState(ac.gState); }// change to G composite state
…..
 }
class B extends TestState {….}

class G extends TestState {
  AbsRegion1State  region1State;
  AbsRegion2State  region2State;
```

```
  void entry() { region1State.entry();
   region2State.entry(); }
  void t4() { region1State.exit();
   region2State.exit(); exit();
   ac.setState(ac.bState); // sets the new state }
  void t2() { region1State.t2(); }
  void t3() { region2State.t3(); }
  void setRegion1(AbsRegion1State subRegion1) {
   region1State = subRegion1; region1State.entry(); }
  void setRegion2(AbsRegion2State subRegion2) {
   region2State = subRegion2; region2State.entry(); }
…..}
class AbsRegion1State{
  Test  m_context;  // super Context object
  G   s_context;  //  sub Context Object
  //  defining abstract methods
}
class C extends AbsRegion1State {
  void t2() { exit();
   // changes to next substate
   s_context.setRegion1(m_context.dState);  }
….}
class D extends AbsRegion1State {
  void entry() {  // implementing join
   if (s_context.region2State.equals(m_context.fState) {
     m_context.t4(); // triggers transition } }
…}
class AbsRegion2State{
    Test  m_context; G  s_context;
    //  defining abstract methods
}
class E extends AbsRegion2State {
  void t3() { exit();
   // changes to next state
   s_context.setRegion2(m_context.fState);  }
…}
class F extends AbsRegion2State {
  void entry() {  // Implementing join
   if (s_context.region1State.equals(m_context.dState) {
    m_context.t4(); // triggers the transition  } }
…};
```

Fork is implemented in the t1()method of the concrete class A. On event *t1*, first of all the exit action of the *A* state is executed then the two concurrent states *C* and *E* are set by calling the fork method of the context Test class and then the state is changed from *A* to composite state *G* and the entry action of the *G* state is executed which in turn executes the entry actions of the concurrent states *C* and *E*.

To implement join we have to make sure that all the multiple source state must be active prior to the firing of the transition. This can be done in the entry() methods of the source D and F classes. If the other state is active then the corresponding event of the super context class is triggered which will delegate it to the current active state G. The t4() first executes the exit actions of the substates followed by the exit action of its own and finally the state is changed to *B* and its entry action is executed.

We also generated code for Figure 5 using Rhapsody in J [14]. A section of code generated by Rhapsody is as follows:

```
class Test implements RiJStateConcept {
   Reactive reactive;
   static final int RiJNonState=0; static final int G=1;
   static final int Region2=2;  static final int F=3;
   static final int E=4;  static final int Region1=5;
   static final int D=6;  static final int C=7;
   static final int B=8;  static final int A=9;
   protected int rootState_substate; protected int rootState_active;
   protected int Region1_substate; protected int Region1_active;
   protected int Region2_substate; protected int Region2_active;
   Test(RiJThread p_thread) {//constructor
     reactive = new Reactive(p_thread); }
   boolean startBehavior() {
     boolean done = false; done = reactive.startBehavior();
     return done }
class Reactive extends RiJStateReactive { // inner class
   Reactive(RiJThread p_thread) { // constructor
     super(p_thread); initStatechart(); }
   boolean  isIn(int state) {
     if (Region1 == state) return isIn(G);
     if (Region1_substate == state) return true;
     if (Region2 == state) return isIn(G);
     if (Region2_substate == state) return true;
     if (rootState_subState == state) return true;
     return true; }
   int rootState_dispatchEvent(short_id) {
int res =RiJStateReactive.TAKE_EVENT_NOT_CONSUMED;
      switch (rootState_active) {
        case A : { res =A_takeEvent(id); break; };
        case G : { res =G_dispatchEvent(id); break; };
        case A : { res =B_takeEvent(id); break; };
        default : break; }; return res; }
   int G_dispatchEvent(short_id) { …. }
   int Region1_dispatchEvent(short_id) {….}
   int Region2_dispatchEvent(short_id) {….}
   protected void initStatechart() {
     rootState_substate = RiJNonState;
     rootState_active = RijNonState;
     Region1_substate = RiJNonState;
     Region1_active = RijNonState;
     Region2_substate = RiJNonState;
     Region2_active = RijNonState; }
……
   int DTaket4() {
int res =RiJStateReactive.TAKE_EVENT_NOT_CONSUMED;
    if (isIn(F)) {
       G_Exit(); B=entDef();
   res = RiJStateReactive.TAKE_EVENT_COMPLETE; }
    return res;  }
    int D_takeEvent(short id) {
int res =RiJStateReactive.TAKE_EVENT_NOT_CONSUMED;
       if (event.isTypeOf(t4.t4_Default_id)) {
          res = DTaket4(); }
if(res==RiJStateReactive.TAKE_EVENT_NOT_CONSUMED)
      {res = Region1_takeEvent(id); }
       return res; }
    void D_entDef() {
       D_enter();     }
    void D_enter() {
       Region1_subState = D;
       Region_active = D;
       DEnter():     }
……
……}
…} // end of class Reactive and class Test
```

```
class t4 extends RiJEvent { // event as class
   static final int t4_Default_id = 18619;
   t4() {   // constructor
      lld = t4_Default_id; }
   boolean  isTypeOf(long  id) {
      if (t4_Default_id == id)     return true;
      else     return false;
   }
} // end of class t4
```

5. Comparison with Rhapsody

Rhapsody [14] is a CASE tool that allows creating UML models for an application and then generates C, C++ or Java code for the application. Code generation in Rhapsody is based on the Object Execution Framework (OXF) [14]. The dynamics of the model are defined in the framework classes and hard-coded in the code generator.

In Rhapsody's model, events are implemented as classes and transition-searching is performed by executing a *switch* statement. Whereas in our approach, events become methods; state hierarchy is implemented by object composition, and transition-searching is automatically performed by using the concept of *polymorphism*. We compared the code generated by our approach and by Rhapsody for Figure 5. Table 1 shows the findings of the comparison. The figures for Rhapsody do not include the code added by the OXF Framework.

	Rhapsody*	Our approach
Source Code: No. of lines	675	250
Source Code: No. of bytes	24270	6420
No. of classes	7	11

Table 1: Comparing the compactness of the generated code

1. Code generated by our approach is more compact. The source code generated by Rhapsody excluding the OXF code is still approximately three times longer than the code generated by our approach, as shown in Table 1.
2. Rhapsody code is difficult to understand. It uses data values to define states and have the operations in the Reactive class check the data explicitly. In such case state transitions implemented as assignments to some variables and have no explicit representation. It puts the transition-selection code in the *switch* statement inside the *takeEvent(short id)* method of the Reactive class. This makes the code difficult to understand. Our code converts each event into an operation call. The appropriate method is selected on the principle of polymorphism. The transition code is put in separate methods in the corresponding class. All the states and transitions are thus explicit without using any conditional statements. This contributes to making the code more readable.
3. Our code is easy to maintain. We put all behavior associated with a particular state into one object. Because all state-specific code is contained in a single class, new states and transitions can be added by defining new classes and operations. In Rhapsody, the actual behavior of the system that was represented as a set of statecharts is buried into the generated code and the OXF framework.
4. Our approach looks like introducing too many small classes, because the behavior for different states is distributed across several states. This increases the number of classes. However, such distribution eliminates large conditional statements. Large conditional statements are undesirable because they tend to make the code less understandable and difficult to modify and extend.

These differences in the mechanisms make the resulting code of our approach compact, more readable and maintainable.

6. Related Work

The most related work is that of Harel and Gery[13] whose tool called Rhapsody [14] generates C++ or Java code from UML models. As shown in the previous section, our code is more readable and maintainable than Rhapsody.

Kohler et al. [15] presented an approach for code generation from statecharts. Their approach adapts the idea generic array based state-table but uses object structure to represent state-table at runtime. They use objects to represent states of a statechart and attributes to hold the entry and exit actions. A library function *handleEvent*() is used to interpret the state-table and to react on events and to issue appropriate action methods. The state table is more complex and expensive to set up. In our approach we have implemented entry and exit action as methods and states as objects. The code of our approach is simple and more efficient.

Tomura et al. [16] presented the statechart design pattern, which define classes and state-transition execution mechanism for realizing the dynamic behavior of device component models of an open distributed control system. Context class represents the class that has the dynamic behavior specified by the statechart. The object of this class has only one StateMachine object. StateMachine is the class for describing the statechart. The objects of this class consist of two sets of states and transition. The objects correspond to either of a statechart diagram itself, sequential substates or concurrent substates. Events guards and actions are also implemented as classes.

Knapp and Merz [17] described a set of tools called Hugo. A generic set of Java classes provides a standard runtime component state for statecharts. States are represented as objects. The *run* method is used to setup and initialize the associated statechart. Hugo code

generation is interpretative in nature and is not producing the optimized code.

Gurp and Bosch [18] presented FSM framework to implement statechart. States, transitions and actions are represented as objects. Similar to the State pattern, there is Context component that has a reference to the current state. Current state is represented as a state object rather than a state subclass. The transition object has a reference to the target state and an Action object. State transition in FSM framework is about twice as expensive as in the State pattern implementation for the simple transition. Transition searching is done by a look up in a hashtable object. The hashtable object maps event names to transition.

7. Conclusion and Future Work

An implementation approach for generating Java code from the UML statechart diagram has been described. By representing states as objects, the concept of object composition and delegation with state design pattern is used to implement the hierarchical states and concurrent states. The proposed approach successfully deals with most of the statechart concepts such as substates, history nodes and compound transitions (fork/join). Some of the statechart concepts such as guards and branches, time and signal events will be implemented in near future. The proposed approach can be used as a basis for automatic code generation for UML statechart diagrams. We are currently working on the implementation of the proposed approach.

Our approach is an object-oriented approach so it can be used to generate low-level code in other object-oriented languages like C++ etc. The code generation engine has to be tailored to the target language as some features are implemented in a different manner in different object-oriented languages.

References

[1] Object Management Group, OMG Unified Modeling Language Specification Version 1.4, OMG, 2001.

[2] D. Harel, Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, No.8, 1987, 231-274.

[3] B.P. Douglass, *Real Time UML – Developing efficient objects for embedded systems* (Massachusetts: Addison-Wesley, 1998).

[4] A. S. Ran, Modeling states as classes, *Proc. Technology of Object-Oriented Languages and Systems Conference*, 1994.

[5] A. Sane, R. Campbell, Object-Oriented state machines: subclassing, composition, delegation, and genericity, *ACM SIGPLAN Notices, OOPSLA'95*, vol.30, Austin, Texas, USA, 1995, 17-32.

[6] K. O. Chow, W. Jia, V. C. P. Chan and J. Cao, Model-based generation of Java code, *Proc. International Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, Las Vegas, USA, 2000.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: elements of reusable object-oriented software* (Massachusetts: Addison Wesley, 1995).

[8] S. M. Yacoub and H. H. Ammar, A pattern language of statecharts, *Proc. Fifth Annual Conf. on the Pattern Languages of Program (PLoP '98)*, Monticello, IL, USA, 1998, TR #WUCS-98-29.

[9] M. Samek and P. Montgomery, State-oriented programming, *Embedded Systems Programming*, August 2000, 22-43.

[10] J. Ali and J. Tanaka, Converting statecharts into Java code, *Proc. Fourth World Conf. on Integrated Design and Process Technology (IDPT'99)*, Dallas, Texas, USA, 2000 (CD-ROM).

[11] J. Ali and J. Tanaka, Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams, *Journal of Computer Science & Information Management (JCSIM), Vol.* 2, No. 1, 2001, 24-34.

[12] J. Ali and J. Tanaka, An object oriented approach to generate executable code from OMT-based dynamic model, *Journal of Integrated Design and Process Science, Vol.* 2, No. 4, 1998, 65-77.

[13] D. Harel and E. Gery, Executable object modeling with statecharts, *Computer, Vol. 30*, No. 7, 1997, 31-42.

[14] Rhapsody case tool reference manual, I-Logix Inc. http://www.ilogix.com.

[15] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf, Integrating UML diagrams for production control systems, *Proc. 22nd International Conf. on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000, 241-251.

[16] T. Tomura, S. Kanai, K. Uehiro and S. Yamamoto, Object-oriented design pattern approach for modeling and simulating open distributed control system, *Proc. IEEE International Conf. on Robotics and Automation (ICRA 2001)*, Seoul, Korea, 2001, 211-216.

[17] A. Knapp and S. Merz, Model checking and code generation for UML state machines and collaborations, *Proc. 5th Workshop on Tools for System Design and Verification*, Reisenburg, Germany, 2002, 59-64.

[18] J. V. Gurp and J. Bosch, On the implementation of finite state machines, *Proc. IASTED International Conf. on Software Engineering and Applications, (SEA'99)*, Scottsdale, AZ, USA, 1999, 172-178.