MAPPING UML STATECHARTS TO JAVA CODE

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 used for modeling the dynamic aspects of systems. The UML statechart diagrams include many concepts that are not present in most popular programming languages, like Java or C++. There exists a gap between high level modeling language and a programming language. There is not a one-to-one mapping between a statechart and its implementation. 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 an approach to generate readable, efficient and compact executable code from the UML statechart diagram in an object-oriented (OO) language like Java using design patterns. By representing states as objects, we extend the state design pattern to implement the hierarchical states using the concept of object composition and delegation. We also propose an approach to implement signal and time events, guards and branches and internal transitions. The proposed approach makes elegant handling of most of the statechart features.

KEY WORDS

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

1. Introduction

The UML [1] is currently the most widespread software modeling language. The emergence of the UML 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 OO design, behavioral modeling aims at describing the behavior of objects using state machines. The UML statechart diagram is a graph that represents a state machine [1]. A UML statechart describes the dynamics of a model element as it changes its internal state as the reaction of receiving some external stimuli. UML statecharts can describe the behavior of a classifier (a class) or a behavioral feature (a method of a class). The semantics and notation used in UML statecharts mainly follow Harel's statecharts [2] with extensions to make them OO. A statechart attached to a class specifies all behavioral aspects of the objects in that class.

The OO methodologies using statecharts describe in detail the steps to be followed for describing the behavior of objects during analysis and design phases. But they fail to show how the analysis and design models of a system shall be converted into implementation code due to lack of syntactic support by the OO programming languages for statecharts. It is difficult for a large fraction of programmers to convert the behavioral models into executable code. The UML statechart diagrams include many concepts that are not present in most popular programming language like Java and C++, e.g. states, fork, events, etc. This means there is not a one-to-one mapping between a statechart and its implementation. Some model elements, like history states, can be implemented in many different ways. This clearly contrasts with class diagrams that often can be easily implemented in a programming language supporting concepts like classes and objects, composition and inheritance. The translation of class diagrams to an OO programming language is easy and provided by most CASE tools.

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 level [3]. This works well for simple statecharts but this solution is not scalable. More complex statechart concepts like hierarchical states, history states, fork, join etc. cause serious problems for this approach as the code becomes complex and is difficult to read and maintain. Ran [4] examined the relation between states and classes and represented states as classes. Sane and Campbell [5] said that states could be represented as classes and transitions as operations. They proposed to reuse behavior in state machines through inheritance of other state machines and implemented embedded states by making a table for the superstate. In the approach of Chow et al. [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, call and signal events. Time events were not implemented.

Design pattern approach [7] is widely used in OO software design. The approach specifies reusable mechanisms for collaboration and interaction among classes or among objects to solve common OO problems in any domain. Several design patterns have been proposed to implement statecharts. *Conditional Statements* pattern [3], *StateTable* pattern [3], *Basic Statechart* pattern [8], Hierarchical State Machine (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.

Our approach is an effort to bridge the gap between design and implementation. We have been working on simple and efficient implementation of UML statecharts in an OO language like Java. Through mapping between UML and Java, we are able to generate low-level java code directly from the statechart diagram. Some of the results of our research, which includes a limited treatment of UML statechart concepts, have already been published [10]. In this paper, we will describe our approach for implementing the hierarchical states in Java by extending State pattern with object composition and delegation. We will also describe our approach for implementing internal transitions, signal events, time events and guards and branches

2 Hierarchical States

We illustrate our approach using an audio cassette player system as an example. Figure 1 shows a statechart for cassette player system having sequential substates.

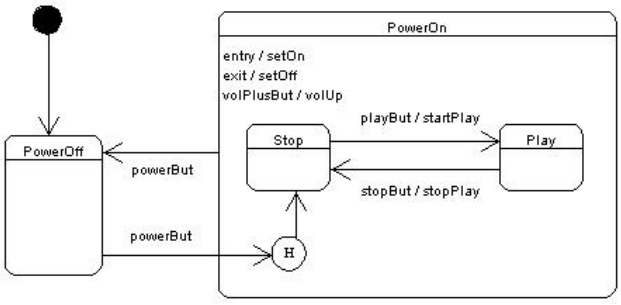

Figure 1. Statechart for cassette player system

The possible events for the system are *powerBut*, *volPlusBut*, *playBut* and *stopBut*. The *volPlusBut* is the internal transition. There are two top-level states PowerOff and PowerOn. These states are activated alternatively whenever a *powerBut* event occurs. The PowerOn state is the composite state with two sequential substates Stop (default) and Play. The *volUp*, *startPlay* and *stopPlay* are the actions on transitions. History state allows the composite state to remember the last substate that was active in it prior to the transition from the composite state.

2.1 Implementation Approach

Our approach for implementing UML statechart diagram is based on [11], [12], [13] and the *State* pattern [7]. State pattern puts all behavior associated with a particular state into one class. The *State* pattern does not deal with the hierarchical states, so some mechanism is needed to implement the state hierarchy. When a composite state is active, exactly one of its sequential substates is active. Transitions affecting the superstate apply at all levels of nesting within that superstate. 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.

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.

To implement a statechart diagram, an OO approach is used. The class, whose behavior is represented by the statechart, becomes the super context class. An abstract state class is generated which defines the interface for encapsulating behavior associated with the states of the statechart. Each state of the statechart becomes a derived class from the abstract state class. Each transition becomes an operation in the corresponding state class in order to provide a uniform and convenient way of invoking some services on the context object. Each action of the statechart becomes a method in the context class. The context class holds the reference of the current active state in the state object and delegates all events for processing to the current state object. State transitions are accomplished by changing the current state object. 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.

The code generated for the cassette player system of Figure 1 is as follows:

```
class CPlayer {
  CPlayerState  state;            // state object
  AbsOnState  onHistory;      // for History state
    // References for all the state objects
  PowerOff  offState;   PowerOn  onState;
  Stop  stopState;       Play  playState;
  CPlayer() { // constructor
    // create state objects only once
    offState = new PowerOff(this); onState = new PowerOn(this);
    stopState= new Stop(this,onState);
    playState = new Play(this,onstate);
    state = offState;            // sets the default state
    onHistory = stopState; // setting the history for the first time  }
```

```
  void setState(CPlayerState st) {        // setting new state
    state = st;
      //sets the most recent active substate of PowerOn state
    if(state.equals(onState)) { onState.substate = onHistory;}
    state.entry();  // executes the entry action    }
// Delegates incoming Events to concrete state class
  void powerBut() {    state.powerBut();   }
  void volPlusBut() {    state.volPlusBut();   }
  void playBut() {    state.playBut();   }
  void stopBut()  {    state.stopBut();   }
// Actions becomes methods in Super Context class
  void setOn() {……}
  void setOff() {……}
  void volUp() {……}
  void startPlay() {……}
  void stopPlay() {……}
}
class CPlayerState {      // Abstract Class
   CPlayer  cp;  // reference to the context object
……..// Declaring abstract methods
}
class PowerOff extends CPlayerState {
    void powerBut() {
      exit();  // executes the exit action of current state
      cp.setState(cp.onState);    // change to PowerOn state  }
}
class PowerOn extends CPlayerState {
  AbsOnState    substate;      // reference for nested statechart
  void entry() {
    substate.entry();  // executes entry action of active substate
    cp.setOn():          // executes entry action    }
  void exit() {
    cp.setoff();         // executes exit action
    cp.onHistory = substate;  // adjusting the history node    }
  void volPlusBut() {      // Internal Transition
    cp.volUp();  // implements action     }
  void playBut()  {        // delegates to nested substate object
    subState.playBut();      }
  void stopBut()  {        // delegates to nested substate object
    subState.stopBut();      }
  void powerBut() {
    substate.exit();  // executes the exit action of nested substate
    exit();    // execute the exit action of the current state
    cp.setState(cp.offState);   // change to PowerOff state    }
  void setSub (AbsOnState sub) {  // setting the active substate
    subState = sub;    substate.entry(); // entry action      }
}
class AbsOnState{ // Abstract Class for Nested Statechart
  CPlayer   m_context;  //  Reference for Super Context Object
  PowerOn  s_context;  //  Reference for Sub Context Object
…….//  defining abstract methods
}
class Stop extends AbsOnState {    // Sequential substate
  void playBut() {
    m_context.startPlay(); // executes the action of Context class
    exit(); // exit action of current substate
   s_context.setSub(m_context.playState);  // sets new substate }
}
class Play extends AbsOnState {……}
```

The *CPlayer* class is the super context class. The *state* object holds the reference of the current active state. The *CPlayer* object delegates all incoming events to its current state object (*state*). All the concrete state objects are created once in the constructor of *CPlayer* class. The *CPlayerState* and *AbsOnState* are the abstract classes providing a common interface for the concrete state classes. The history state is implemented by providing a reference *onHistory* in the *CPlayer* class, which sets the *onHistory* at the start to Stop (default) state and later on it is adjusted to the current active substate in the exit() method by the PowerOn composite state. When PowerOn state becomes active, its most recent active substate is also set. The concrete state classes handle the events. First of all the action associated with the event is executed followed by the exit action of the current state and finally the setState() method of the CPlayer class is called to set the new state. If the new state is PowerOn then the last active substate is also set. PowerOn class becomes the context for the nested statechart. The *subState* object keeps the reference of the current active substate. If the target of a transition is the composite state then it is handled by the composite state but if the target is the nested state then it will be delegated to the current active substate. An internal transition executes without exiting and re-entering the state in which it is defined. The internal transition volPlusBut is implemented in the corresponding *PowerOn* class. In this case only the action associated with the internal transition is executed and the exit action and the setSub() methods are not called.

3 Time and Signal Events

3.1 Time Events

A time event is an event that represents the passage of time. It is specified with the keyword *after* followed by some expression that evaluates to a time period. The time is normally counted since the state is entered. Figure 2 shows statechart of the cassette player system having a transition on time event.

Figure 2. Statechart having a time event

To implement timeout events, we have developed a simple *Timer* class, which can be used by any state object. The *Timer* class has an integer variable representing the number of milliseconds and a reference to the state object for which a *Timer* class object is created. These two variables are set when a timer object (an instance of *Timer*) is newly created. The timer sends a timeout() message to the state object when the specified number of milliseconds has elapsed. There is a *TimerState* interface that has timeout() method. The state class uses a maximum priority thread so that it can send the timeout() message as soon as the time is expired. Before the time is expired, the thread is in sleep state so it does not effect the

usual execution of the system. Following is the Java code that implements the statechart of Figure 2.

```
interface TimedState { void timeout();   }
class Timer extends Thread {
   int millisec; TimedState state;
   Timer (TimedState s, int ms) { // constructor
      state = s;
      millisec = ms;
      setPriority(Thread.MAX_PRIORITY);}
   void run() { // goes to sleep until the time is expired
      try{ sleep(millisec);
      catch(InterruptedException e) {
   // send timeout message to state
      state.timeout(); }
}
class CassetteEnd extends CPlayerState
implements TimedState {
void entry() { // sets a new timer to 5 secs upon entry
   timer = new Timer(this,5000);
   timer.start(); }
void timeout() { //called from the timer
   cp.setState(cp.stopState); }
}
```

3.2 Signal Event

A signal represent a named object that is dispatched asynchronously by one object and then received by another. Following the UML semantics, our approach assumes an event queue and an event dispatcher mechanism maintained by the system. In the case of signal events, the sender object does not call directly an operation of the receiver object. Instead, the sender places the event in an event queue maintained by the system. Control remains in the sender object. An event dispatcher, which runs in a separate thread, dispatches the events from the event queue to the specified objects one by one.

As long as an object is the receiver of events, there is nothing to do special in its implementation. However, while responding to some event, if an object sends messages to other objects, then it needs to differentiate calls and signal events In the case of a call, a method in the receiver object will have to be called using a reference to that object in the sender object. In the case of a signal, the method name and the receiving object reference will have to be placed in the system's event queue.

4 Guards and Branches

In this section, we will describe our approach for implementing guards and branches. A guard is a Boolean condition that returns a TRUE or FALSE value that controls whether or not a transition is taken following the receipt of a triggering event. A transition with a guard is only taken if the triggering event occurs and the guard evaluates to TRUE. A guard should not have side effects. The conditional branch or choice splits an incoming transition into several disjoint outgoing transitions. Each outgoing transition has a guard condition that is evaluated after prior actions on the incoming path have been completed.

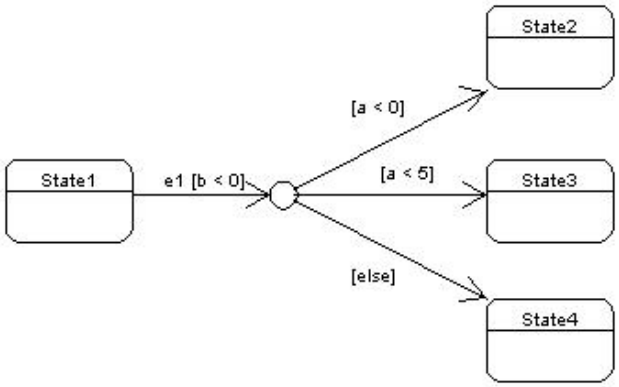

Figure 3. Statechart with branches and guard conditions

Implementing the branch and guard condition is straightforward. All the code for checking the branch and guard condition is put inside the *if* statement of the corresponding event method. The method is called when the corresponding event occurs while the source state is active. Following is the part of Java code for the statechart of Figure 3.

```
class state1 extends AbsState {
  void  e1() {
      if (b < 5 ) {
           if (a < 0 ) {
              exit();   // executes the exit action
              ac.setState(ac.s2State); //   sets the new state     }
           else  if  (a < 5) {
                     exit();
                     ac.setState(ac.s3State); }
                else
                     exit();
                     ac.setState(ac.s4State);}
  }
```

5. Comparison with Rhapsody

The most related work is that of Harel and Gery [14] and their supporting tool Rhapsody [15]. Rhapsody 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) [15]. The tool does not optimize the generated code and the dynamics of the model are defined in the framework classes and hard-coded in the code generator. The code generator automatically derives model classes from the framework classes based on the application classes.

Rhapsody uses data values to define states and the operations in the *Reactive* (Context) class check the data explicitly. Rhapsody represents events as classes. The state accepts a given signal event via the *gen()* operation, which queues the event in its associated manager. The

manager later injects it to the *Reactive* instance for consumption. The state transitions are implemented as assignments to some variables and have no explicit representation. The transition-searching is performed by executing a switch statement in the *Reactive* class. The handling of time events is buried in OXF.

In our approach, states become classes and state hierarchy is implemented by object composition. Our code converts each event into an operation call and transition-searching is automatically performed by using the concept of *polymorphism.* The transition code is put in separate methods in the corresponding classes. All the states and transitions are thus made explicit without using any conditional statements.

5.1 Comparison of Generated Code

We compared the code generated by our approach and by Rhapsody for Figure 1. Table 1 shows the findings of the comparison. The figures for Rhapsody do not include the code added by the OXF Framework*.

Table 1: Comparing the compactness of the generated code

	Rhapsody*	Our approach
Source Code: No. of lines	620	220
Source Code: No. of bytes	17780	6070
No. of classes	6	7

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 whereas we have implemented states as classes. In Rhapsody, the transition-searching is performed by executing a switch statement whereas in our approach, it is performed by using the concept of *polymorphism.* All the states and transitions are thus explicit without using any conditional statements. This contributes to making our code more readable.
3. Our code is easy to maintain. 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. We have put all behavior associated with a particular state into one object. As all the state-specific code is contained in a single class, new states and transitions can be added easily by defining new classes and operations.

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

6. Related Work

Kohler et al. [16] presented an approach for code generation from statecharts. Their approach adapts the idea of generic array based state-table but uses an object – oriented implementation of the state-table at runtime. The states of the statecharts are subclasses from *FReactive* class. The *FReactive* provides a pointer to the current state of the reactive object and an abstract initStatechart() and handleOneEvent() method. The initStatechart method is used to create the state-table, Each reactive object has its own event queue inherited from *FReactive* class. The handleOneEvent() method is used to interpret the state-table and to react on events and to issue appropriate action methods. They have employed more than one event queue. They did not discuss the history states and timeout events implementation. The state table is more complex and expensive to set up. In our approach entry and exit action are implemented as methods and states as objects.

Tomura et al. [17] 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, entry and actions, and guards are also implemented as classes. They did not discuss the signal and timeout events.

Knapp and Merz [18] described a set of tools called Hugo. A generic set of Java classes provides a standard runtime component state for statecharts. The run() method is used to setup and initialize the associated statechart. Every state of a statechart is represented by a separate object that provides methods for activation, deactivation, initialization and event handling. Hugo code generation is interpretative in nature and is not producing the optimized code. The time events and history states are not implemented.

Gurp and Bosch [19] presented Finite State Machines (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. Time events and history nodes are not implemented.

7. Conclusion and Future Work

An OO approach for generating Java code from the UML statechart diagram has been described. By representing states as objects, we have used the concept of object composition and delegation with state design pattern to implement the hierarchical states. The states are represented as classes and transitions as operations, thus eliminating the need of large conditional statements. This makes the components of the statechart diagram explicit and the resulting code easier to understand and maintain. It is easier to add new states and transitions. The proposed approach successfully deals with most of the statechart concepts such as hierarchical states, internal transitions, call, signal and time events, guards and branches. The code generated by our approach is approximately three times more compact than Rhapsody. 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 to verify the research results.

Our approach is an OO approach and in the present study we have used Java language as the target language. However, our approach is general so it can be used to generate low-level code in other OO languages like C++. The code generation engine has to be tailored to the target language as some features are implemented differently in different OO programming languages.

References

[1] Object Management Group (OMG), Unified Modeling Language (UML) Specification Version 1.5, OMG, 2003. http://www.omg.org/technology/documents/formal/uml.htm

[2] D. Harel, Statecharts: A visual formalism for complex systems, *Science of Computer Programming*, 8(3), 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*, 13(8), 2000, 22-43.

[10] I.A. Niaz and J. Tanaka, Code generation from UML statecharts, *Proc. 7th IASTED International Conf. on Software Engineering and Applications (SEA 2003)*, Marina Del Rey, USA, 2003, 315-321.

[11] 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).

[12] J. Ali and J. Tanaka, Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams, *Journal of Computer Science & Information Management (JCSIM)*, 2(1), 2001, 24-34.

[13] 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*, 2(4), 1998, 65-77.

[14] D. Harel and E. Gery, Executable object modeling with statecharts, *Computer, 30*(7), 1997, 31-42.

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

[16] 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.

[17] T. Tomura, S. Kanai, K. Uehiro and S. Yamamoto, Developing simulation models of open distributed control system by using object-oriented structural and behavioral patterns, *Proc.4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Madgeburg, Germany, 2001, 428-437.

[18] 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.

[19] 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.