Converting Statecharts into Java Code

Jauhar Ali and Jiro Tanaka

Institute of Information Sciences and Electronics
University o f Tsukuba
Tennodai 1-1-1, Tsukuba, Ibaraki 305-8573 Japan
jauhar@softlab.is.tsukuba.ac.jp and jiro@.is.tsukuba.ac.jp

ABSTRACT

This paper presents an implementation model to convert statecharts, representing the behavior of multi-state classes in a system, into executable code in an object-oriented language like Java. The concept of a helper object is introduced which handles all the state-specific requests forwarded to it by the multi-state domain object. A new helper object replaces the old one, whenever the state of the domain object changes. The proposed model follows the object variant of statecharts supported by the Unified Modeling Language (UML). Our model can work as a basis for automatic code generation for object-oriented systems.

Keywords

Statecharts, OOA/D, Implementing object behavior, Java

1 INTRODUCTION

Statecharts (Harel, 1987, 1988; Harel and Naamad, 1996), originally designed for modeling reactive systems, are used by most of the current object-oriented methodologies (Rumbaugh et al., 1991; Jacobson, 1992; Booch, 1991; Desfray, 1994; Rational Software Corporation, UML) to describe the dynamic behavior of a system. Statecharts are characterized by a number of conceptual shortcuts, such as hierarchical states, concurrent states, and history and branch nodes, which, in combination, achieve a significant compaction of specifications over most other state-based formalisms.

A statechart attached to a class specifies all behavioral aspects of the objects in that class. Some classes do not need statecharts because their objects are uni-state, and they always behave in the same way.

To implement the classes of an object-oriented system, one has to implement the corresponding statecharts which specify the behavior of the classes. A large fraction of programmers, however, have difficulty in converting statecharts into executable code because most programming languages do not provide syntactic support for statecharts.

We have been working on implementing the dynamic behavior of an object-oriented system. Some of the results of our research, which includes a limited treatment of state transition diagrams, have already been published (Ali and Tanaka, 1996, 1997, 1998a, 1998b). The present study focuses on implementing statecharts in general and addresses all the important components of statecharts, such as, hierarchical states, concurrent states, history, fork, join and branch nodes. We show how statecharts representing the behavior of a class can be converted to executable code in Java (Gosling et al., 1996) language. For notation and semantics, we refer to Unified Modeling Language (Rational Software Corporation) which has made few small modifications in the original statecharts to make it fit into the object-oriented paradigm.

2 STATES AS OBJECTS

Implementing uni-state classes of objects is straightforward because they always respond to other objects in the same way. Multi-state objects are difficult to implement because each time they receive some external message, they respond differently depending on the states they are currently in.

Our investigation shows that the implementation of a multi-state class of objects can be made simple if we arrange a helper object that encapsulates all the state-specific behavior of the multi-state domain object. The helper object represents the current state of the domain object and implements the behavior specific to the current state. The domain object delegates all external messages to its helper object, and the helper object responds to the messages on behalf of the domain object. Since the helper object represents only one state of the domain object, its methods do not contain any conditional statements and are simple to implement. When the state of the domain object changes, a new helper object, implementing the behavior specific to the new state, replaces the old one. The

reference to the helper object is identified as cs (current state) in the domain class.

We use a cassette player system as an example to demonstrate our approach. Figure 1 shows a simple statechart for the cassette player having two states: *Stop* and *Play*. In the *Stop* state, when a *playBut* event occurs, the device executes the *startPlay* action and goes to the *Play* state. In the *Play* state, when a *stopBut* event occurs, the device executes the *stopPlay* action and goes to the *Stop* state. This behavior can be implemented in Java as follows.

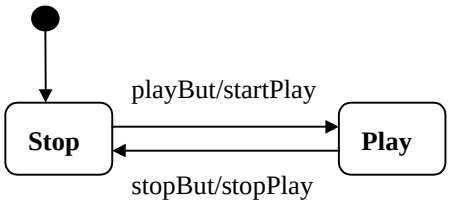

Figure 1: Statechart for a cassette player

```
class CassettePlayer {
  CassettePlayerState cs; //helper object
  CassettePlayer(){cs = new Stop(this);}
  void stopBut(){cs.stopBut();}
  void playBut(){cs.playBut();}
  void startPlay(){...}
  void stopPlay(){...}
}
class CassettePlayerState {//abstract class
  CassettePlayer context; //reference to domain object
  CassettePlayerState(CassettePlayer c){//constructor
    context=c;
  }
  void stopBut(){}
  void playBut(){}
}
class Stop extends CassettePlayerState {
  Stop(CassettePlayer c){super(c);}
  void playBut(){context.startPlay();
  context.cs = new Play(context);}
}
class Play extends CassettePlayerState {
  Play(CassettePlayer c){super(c);}
  void stopBut(){context.stopPlay();
  context.cs = new Stop(context);}
}
```

The Stop and Play classes implement the behavior specific to the *Stop* and *Play* states respectively. They are subclassed from the CassettePlayerState class which provides a common interface for all state classes. We call these classes as state classes because they implement individual states of the statechart attached to the domain class. All the state classes have a reference to the domain object identified as context. In short, each state in the statechart becomes a class and each transition from that state becomes a method in the corresponding class. All actions become methods in the domain class. The domain object forwards all incoming messages (events) to its current state object (cs).

Internal Transitions, Entry and Exit Actions

A state can have *internal transitions*, *entry* action and/or *exit* action. An internal transition is specified inside a state by an event name, followed by a slash (/), followed by an action name. The action is performed when the state is occupied and the event occurs. Internal transition does not cause the state to change. An entry action is performed whenever the state is entered. It is specified inside the state by the *entry* keyword, followed by a slash (/), followed by the action name. An exit action is performed whenever the state is exited. It is specified inside the state as the *exit* keyword, followed by a slash (/), followed by the action name. Figure 2 shows the statechart for the cassette player having an internal transition and the *entry* and *exit* actions in the *Play* state.

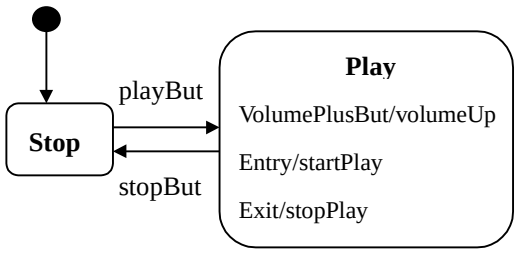

Figure 2: Statechart having entry, exit actions and internal transition

While executing a transition, instantiating new state objects takes time. To make the implementation code more efficient, an instance of each of the state classes can be created beforehand in the domain class. When a transition is fired, an appropriate instance, representing the new state, is assigned to cs instead of instantiating a new object. Using this optimization technique, Java code for the statechart of Figure 2 can be written as follows.

```
class CassettePlayer {
  CassettePlayerState cs; //helper object
  Stop stopState;// reference to Stop state object
  Play playState;// reference to Play state object
  CassettePlayer(){
    // create state objects only once
    stopState = new Stop(this);
    playState = new Play(this);
    cs = stopState;// set default state
    ...}
}
```

```
class Play extends CassettePlayerState {
  void entry(){context.startPlay();}
  void exit(){context.stopPlay();}
  void volumePlusBut(){context.volumeUp();}
  void stopBut(){
    exit();// call the exit action
    context.cs = context.stopState;// set new state
    context.cs.entry();// call the entry action
  }
}
```

As the above code shows, entry and exit actions become entry() and exit() methods in the corresponding class (Play). The CassettePlayerState class should have empty methods for entry() and exit(). A method implementing a transition (e.g., stopBut()) first calls the exit() of the current state, then updates the state, and finally calls the entry() of the new state.

3 COMPOSITE STATES

A composite state can be decomposed using OR-relationships into mutually exclusive disjoint substates or using AND-relationships into concurrent substates. The substates inherit the transitions of the composite state. In the case of OR-type substates, only one of the substates can be active at a given time. In the case of AND-type substates, all the substates become active simultaneously whenever the composite state becomes active. In this section, we show how our statechart implementation model deals with the two types of state hierarchy.

OR-Type Substates and History Nodes

In a state hierarchy, the superstate has transitions that are common to its substates. Whereas in an object-oriented system, a superclass has methods which are common to its subclasses. The similarity between state hierarchy and class hierarchy has led us to represent states as objects and exploit the concept of class inheritance for implementing state hierarchy.

In our approach, where the state-specific behavior of a multi-state domain class are implemented as separate classes (state classes), substates become subclasses of the class for the superstate. The superclass implements the behavior specific to the superstate and the subclasses implement the behavior specific to the substates. As an example, Figure 3 shows a more detailed statechart containing state hierarchy for the cassette player. There is a transition from *PowerOn* to *PowerOff* on the *powerBut* event. This transition is inherited by the two substates: *Stop* and *Play*. There is a transition from the *PowerOff* state to the *history* node inside the *PowerOn* state. This means that the most recent active substate of *PowerOn* prior to the entry will be entered. If the transition is fired for the first time, the *Stop* substate will be entered because a transition goes from the history node to the *Stop* substate. The following Java code implements the statechart of Figure 3.

```
class CassettePlayer {
  //This reference represents the most
  //recent active substate of PowerOn
  PowerOn powerOnHistory;
  CassettePlayer(){
    cs = powerOffState;
    powerOnHistory = stopState;}
  ...
}
class PowerOff extends CassettePlayerState {
  void powerBut(){
  context.cs = context.powerOnHistory;}
}
class PowerOn extends CassettePlayerState {
  void exit(){
  //adjust the history node
  context.powerOnHistory = context.cs;}
  void powerBut(){exit();
  context.cs = context.powerOffState;}
}
class Stop extends PowerOn {... }
class Play extends PowerOn {... }
```

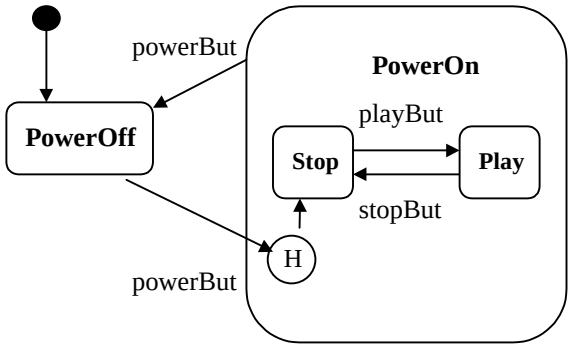

Figure 3: Statechart having state hierarchy

As can be seen in the code, the powerBut() method is implemented in the PowerOn class and is inherited by its subclasses: Stop and Play. The history node inside an OR-type composite state is implemented by providing a reference in the domain class. The reference, which represents the most recent active substate, is updated each time the composite state is exited.

AND-Type substates

When a composite state contains AND-type substates (concurrent substates), the substates become active simultaneously whenever the composite state becomes active. This means that we need to have the helper object (cs), representing the currently active composite state, have references to other objects representing the substates. The helper object can be thought as a composite object containing other (substates) objects. That is why, we

implement the superstate of AND-substates as a composite class that has references to objects of the classes implementing the substates. Figure 4 shows another version of the statechart for the cassette player containing concurrent states. As can be seen in the figure, *Speaker* and *Player*, though concurrent substates of *PowerOn*, are abstract states having other concrete substates. When *PowerOn* is active, both the regions of *Speaker* and *Player* are active too. However, either *Left* or *Right* is active in the *Speaker* region, and either *Stop* or *Play* is active in the *Player* region. Java code for this statechart can be written as follows.

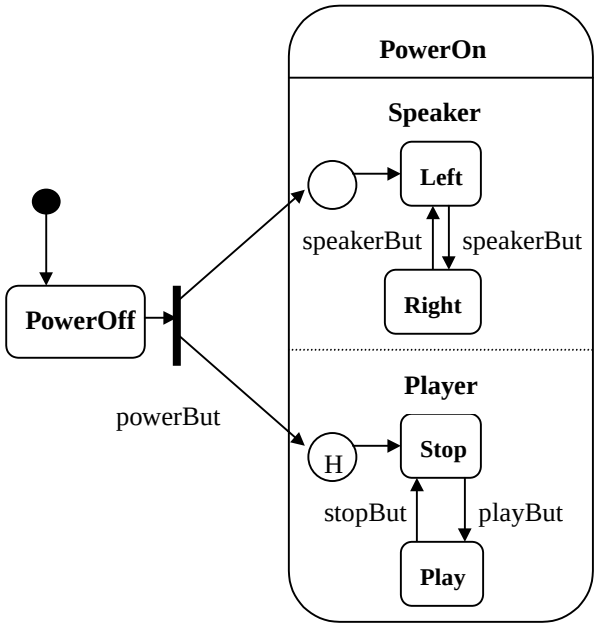

Figure 4: Statechart having concurrent states

```
class CassettePlayer {
  Speaker speakerHistory;
  Player playerHistory;
  CassettePlayerState cs;
  CassettePlayer(){
    cs = powerOffState;
    speakerHistory = leftState;
    playerHistory = stopState;}
}
class PowerOff extends CassettePlayerState {
  void powerBut(){
    //PowerOn becomes active
    context.cs = context.powerOnState;
    //enter the most recent active substates in
    // both regions: Speaker and Player
    ((PowerOn) context.cs).sp =
    context.speakerHistory;
    ((PowerOn) context.cs).pl =
    context.playerHistory;}
}
class PowerOn extends CassettePlayerState {
  Speaker sp;
  Player pl;
  void exit(){
    //updates the most recent active substate
    context.speakerHistory = sp;
    context.playerHistory = pl;}
  void powerBut(){exit();
    context.cs = context.powerOffState;}
  // forwards messages to components
  void speakerBut(){sp.speakerBut();}
  void playBut(){pl.playBut();}
  void stopBut(){pl.stopBut();}
}
class Speaker {void speakerBut(){}
class Player {...}
class Left extends Speaker {
  void speakerBut(){
  ((PowerOn) context.cs).sp = context.rightState;}
}
class Right extends Speaker {...}
class Stop extends Player {... }
class Play extends Player {... }
```

As can be seen in the implementation code, Speaker and Player classes just provide common interfaces to their subclasses. The composite class (PowerOn) has two references: sp and pl, representing the two sub-regions inside the *PowerOn* state. To access these references, we need to cast explicitly context.cs to PowerOn because the type of cs is CassettePlayerState which does not contain a declaration of the references. PowerOn forwards the requests (events) on which there are transitions within the AND-substates to the corresponding substate objects. Transitions from the composite state (e.g., *powerBut*) are implemented in the composite class and are not forwarded to the substate objects. The code also includes implementation of the history nodes inside the two regions.

4 COMPOUND TRANSITIONS

A compound transition may have multiple source states and target states. A compound transition is enabled when all of 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. Arrows come from the source states to the bar and go out from the bar to the target states. The bar may have a trigger (event), but the individual segments do not have any trigger.

Fork

A fork has a single source state and multiple target states. Figure 5 shows a statechart having a fork. The trigger event for the fork is *start*. This means that when the current state is *Setup*, and the event *start* occurs,

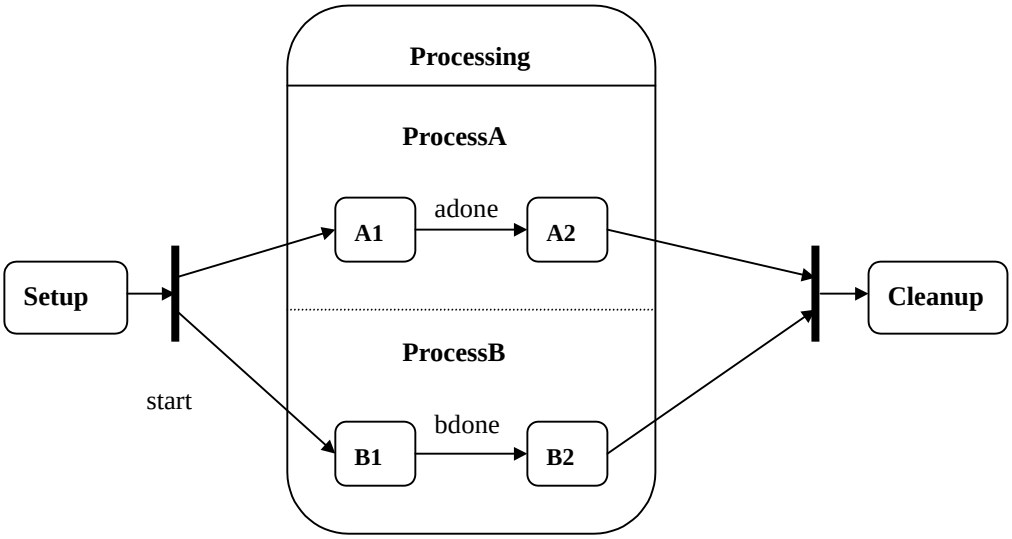

Figure 5: Statechart having composite transitions (*fork* and *join*)

both *A1* and *B1* will become active.

Implementing a fork is straightforward. The method in the source state class, which implements the fork, makes all the targets states active, rather than just one state.

Join

A join (also called a synchronization) has multiple source states and a single target state. Figure 5 also contains a join, going from *A2* and *B2* to *Cleanup*.

Implementing a join needs some thought. There are many source states which should all be occupied to fire the transition. Whenever one of the source state is occupied, we need to check whether all of the remaining source states are also occupied. If the result is true, the transition should be fired. Java code for the fork and join of Figure 5 can be written as follows. We suppose that the domain class name is Device. As shown in the code, the fork is implemented by the start() method in the Setup class. The join is implemented by the entry() methods in the A2 and B2 classes.

```
class Device {
  DeviceState cs;
  Setup setupState; Processing processingState;
  A1 a1State; A2 a2State; B1 b1State; B2 b2State;
  Cleanup cleanupState;
  ...
}
class DeviceState{//empty method declarations}
class Setup extends DeviceState {
  void start(){//implementing fork
  context.cs = context.processingState;
  ((Processing) context.cs).a = context.a1State;
  ((Processing) context.cs).b = context.b1State;}
}
class Processing extends DeviceState {
  ProcessA a;
  ProcessB b;
  void adone(){a.adone();}
  void bdone(){b.bdone();}
}
class ProcessA {//empty method declarations}
class ProcessB {//empty method declarations}
class A1 extends ProcessA {
  void adone(){
    ((Processing) context.cs).a = context.a2State;
    ((Processing) context.cs).a.entry();}
}
class A2 extends ProcessA {
  void entry(){//implementing join
    if (((Processing) context.cs).b instanceof B2){
      context.cs = context.cleanupState;}}
}
class B1 extends ProcessB {
  void bdone(){
    ((Processing) context.cs).b = context.b2State;
    ((Processing) context.cs).b.entry();}
}
class B2 extends ProcessB {
  void entry(){//implementing join
    if (((Processing) context.cs).a instanceof A2){
      context.cs = context.cleanupState;}}
}
class Cleanup extends DeviceState {...}
```

5 GUARDS AND BRANCHES

A guard condition is a boolean expression that may be attached to a transition in order to determine whether that transition is enabled or not. The transition is fired only when the condition is true at the time the trigger event occurs. The boolean expression is written in terms of parameters of the triggering event and attributes of the domain object.

A simple transition may be extended to include a tree structure of branches. Each branch has a guard condition. Only one of the branch is fired at a given time. When the trigger event occurs, if a branch has its guard condition true, it is fired. If no branch has a true condition, the event is ignored. Figure 6 shows the statechart of a telephone system having a transition with guard conditions and branches forming a tree.

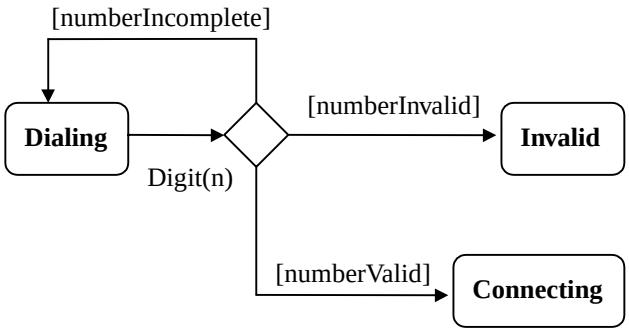

Figure 6: Statechart of a telephone system having branches and guard conditions

To implementing a transition with guard condition, we put all the code inside an *if* statement that checks the condition. The method is called whenever the corresponding event occurs while the source state is active. But the actual transition code is executed only if the guard condition is true. Following is part of the Java code for the statechart of Figure 6.

```
class Dialing extends PhoneState{
  void digit(int n){
    if (numberIncomplete()){
      //code to be in the Dialing State}
    else if (numberValid()){
      //code to go into Connecting State}
    else if (numberInvalid()){
    //code to go into Invalid State}}
}
```

6 MORE ABOUT EVENTS

Timeout Events

A timeout event results from the expiration of some deadline. It is specified by the keyword *after* followed by an expression that evaluates to an amount of time. The time is normally counted since the state is entered. Figure 7 shows another version of the cassette player system having a transition on a timeout event.

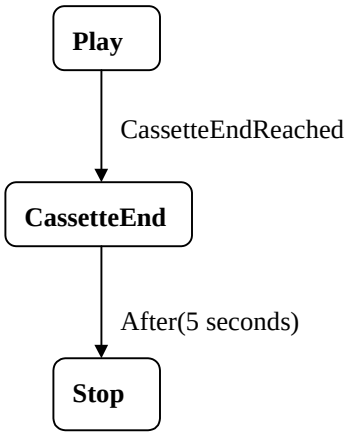

Figure 7: Statechart having a time event

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 have elapsed. There is a TimeState interface that has timeout() method. The state class should implement this interface in order to use the Timer class. A state class that has a transition on a timeout event keeps an object of the Timer class. The Timer 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 7.

```
interface TimedState {void timeout();}
class Timer extends Thread {
  int millisec; TimedState state;
  //constructor
  Timer (TimedState s, int ms) {
    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 CassettePlayerState
  implements TimedState {
  void entry(){
    // set a new timer to 5 secs upon entry
    timer = new Timer(this,5000);timer.start();}
  void timeout(){//called from the timer
    context.cs = context.stopState;}
}
```

Calls and Signals

Until now, the discussion was about events (external messages) received by an object and its response to those events on the basis of its current state. The events can be the result of an operation call by some object, in which case they are called *call events*. An event can also come from the system's event dispatcher, in which case it is a *signal event*. Following the UML semantics, our implementation model assumes an event queue and an event dispatcher maintained by the system. Call events are simple to implement. The sender object just calls an operation in the receiver object. This becomes an event for the receiver object which responds to the event. There should be a method in the receiver object's class to implement the operation. When the operation is called, control is transferred from the sender object to the receiver object. After executing the operation, control is transferred back to the sender object.

In the case of signal events, however, the sender object does not call directly an operation of the receiver object. Instead, the sender places the event (operation call for a particular object) in an event queue maintained by the system. Control remains in the sender object. An event dispatcher, maintained also by the system, runs in a separate thread dispatches the events from the event queue to the specified objects one by one.

As for as an object is the receiver of events, we need nothing to do special in its implementation. However, while responding to some event, if an object sends messages to other objects, we need to differentiate calls and signals. 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.

7 DISCUSSION

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 easily by defining new classes and operations. An alternative would be to use data values to define internal states and have the operations in the domain class check the data explicitly. In such case, state transitions are implemented as assignments to some variables and have no explicit representation. This may be good for efficiency but the actual behavior of the system that was represented as a set of statecharts is buried into the code. It is very difficult to understand the behavior of the system by looking only at the code. Representing different states as separate objects makes the transitions more explicit and the code more understandable. This keeps the flavor of statecharts in the implementation code and is also very helpful for reverse engineering the code back into statecharts.

Our approach may look like introducing too many classes, because the behavior for different states is distributed across several state classes. This increases the number of classes and the implementation of behavior is less compact than a single class. 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.

8 RELATED WORK

The most related work is that of Harel and Gery (1996, 1997), whose tool called Rhapsody (I-Logix Inc.) generates C++ code from statecharts attached to classes in object diagram. The papers reporting the tool do not explain well the generated code, but their approach can be understood by looking at the actual code generated by Rhapsody. The similarity between our model and Rhapsody's model is that both implement the states of a statechart as objects. The differences lie in the treatment of state hierarchy, events and transitions. In Rhapsody's model, events are implemented as classes; state hierarchy is implemented by having pointers to super/sub-state classes; and transition searching is performed by executing a *switch* statement. Whereas in our model, events become methods; state hierarchy is implemented by inheritance; and transition searching is automatically performed by using the concept of *polymorphism*. These differences in the mechanisms make the resulting code of our model simpler, more compact and efficient.

Our mechanism of converting a statechart into implementation code has some similarity with the State pattern (Gamma et al., 1995), but State pattern neither addresses the issue of state hierarchy nor does it address concurrency within state diagrams.

The relation between states and classes is examined by Ran (1994). Sane and Campbell (1995) say that states can be represented as classes and transitions as operations. They implement embedded states by making a table for the superstate and do not consider concurrent states.

9 CONCLUSIONS

An implementation model for statecharts, representing the behavior of multi-state classes, has been described. The introduction of helper objects, greatly simplifies the implementation of multi-state classes of objects. By representing states as objects, the concepts of inheritance and object composition can easily be used to implement hierarchical states and concurrent states. The proposed model successfully deals with almost all statechart concepts, which include history nodes, branches, compound (fork/join) transitions, and timeout, call and signal events. The proposed model can be used as a basis for automatic code generation for an object-oriented system.

REFERENCES

Ali J. and Tanaka J., 1996, “Automatic Code Generation from the OMT-based Dynamic Model,” Proceedings of the Second World Conference on Integrated Design and Process Technology, vol.1, pp. 407-414, Austin, Texas, USA.

Ali J. and Tanaka J., 1997, “Generating Executable Code from the Dynamic Model of OMT with Concurrency,” Proceedings of the IASTED International Conference on Software Engineering (SE'97), pp.291-297, San Francisco, California, USA.

Ali J. and Tanaka J., 1998a, “Implementation of the Dynamic Behavior of Object Oriented System,” Proceedings of the Third World Conference on Integrated Design and Process Technology, pp.281-288, Berlin, Germany.

Ali J. and Tanaka J., 1998b, “Implementing the Dynamic Behavior Represented as Multiple State Diagrams and Activity Diagrams,” Proceedings of AoM/IAoM 16th Annual International Conference, pp.61-68, Chicago, USA.

Booch G., 1991, “Object Oriented Design with Applications,” Benjamin/Cummings, Redwood, California, USA.

Desfray P., 1994, “Object Engineering: The Fourth Dimension,” Addison Wesley, Reading, Massachusetts, USA.

Gamma E., Helm R., Johnson R., and Vlissides J., 1995, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison Wesley, Reading, Massachusetts, USA.

Gosling J., Joy B., and Steele G., 1996, “The Java Language Specification,” Addison Wesley, Reading, Massachusetts, USA.

Harel D., 1987, “Statecharts: A Visual Formalism for Complex Systems,” Science of Computer Programming,” no.8, pp.231-274.

Harel D., 1988, “On Visual Formalisms,” Communications of the ACM, vol.31(5), pp.514-530.

Harel D., and Gery E., 1996, “Executable Object Modeling with Statecharts,” Proceedings of 18th International Conference on Software Engineering, pp.246-257.

Harel D., and Gery E., 1997, “Executable Object Modeling with Statecharts,” Computer, vol.30(7), pp.31-42.

Harel D., and Naamad A., 1996, “The STATEMATE Semantics of Statecharts,” ACM Transactions on Software Engineering and Methodology, vol.5(4), pp.293-333.

i-Logix Inc., “Rhapsody,” http://www.ilogix.com.

Jacobson I., 1992, “Object-Oriented Software Engineering: A Use Case Driven Approach,” Addison Wesley, Reading, Massachusetts, USA.

Ran A. S., 1994, “Modeling States as Classes,” Proceedings of the Technology of Object-Oriented Languages and Systems Conference.

Rational Software Corporation, “ Unified Modeling Language (UML),” http://www.rational.com.

Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., 1991, “Object-Oriented Modeling and Design,” Prentice Hall, Eaglewood Cliffs, New Jersey, USA.

Sane A., and Campbell R., 1995, “Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity,” ACM SIGPLAN Notices, OOPSLA'95, vol.30, pp.17-32, Austin, Texas, USA.