An Object-Oriented Approach To Generate Java Code From UML Statecharts

Iftikhar Azim Niaz and Jiro Tanaka *
University of Tsukuba, Japan

Abstract

The Unified Modeling Language (UML) statechart diagram is used for modeling the dynamic behavior of a system. This paper describes an object-oriented (OO) approach to generate compact and efficient Java code from UML statechart diagrams. The states are represented as objects and all the behavior associated with a state is contained in one object. This localizes the state-specific behavior and partitions behavior for different states. Introducing separate objects for different states makes the transitions more explicit. We have implemented the statechart diagram having sequential and concurrent substates by extending the state design pattern using the concept of object composition and delegation. The method has been successfully implemented in our automatic code generating system, JCode, which generates Java code after reading the specifications of the UML statechart diagram. The paper also presents the results of the experiment in which the code generated by JCode is compared to that of Rhapsody and OCode. The results show that the code generated by JCode is 68% more efficient and four times more compact than that of Rhapsody and 50% more efficient than that of OCode.

Keywords: Code generation, object-oriented analysis and design, CASE, statecharts, object composition, Java.

1. Introduction

The UML [1] is a general-purpose visual modeling language that is used to specify, visualize, construct and document the artifacts of a software system. The emergence of 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. It provides several diagram types that can be used to view and model the software system from different perspectives and/or at different levels of abstraction. In UML based OO design, behavioral modeling aims at describing the behavior of objects using state machines. A state machine is a behavior that specifies the sequence of states an object goes through during its lifetime in response to events [2]. A state machine models the possible life histories of an object of a class. State machines are used for specifying the full dynamic behavior of a single class of objects. The UML statechart diagram visualizes a state machine [2]. A statechart diagram contains states, transitions, events and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of a class and emphasize the event-ordered behavior of an object, which is particularly useful in modeling reactive systems. A statechart attached to a class specifies all behavioral aspects of the objects in that class. The semantics and notations used in UML statecharts mainly follow Harel's statecharts [3] with extensions to make them OO [1].

The OO software is a collection of interacting objects. Benefits of high-level modeling and analysis are significantly enhanced if code can be generated automatically from a model such that the correspondence between the model and code is precisely understood. Model-based code generation produces application code automatically from graphical models of system objects and behavior. Development tools are moving to model-based development to raise the level of abstraction at which developers can work. OO methods help developers analyze and understand a system, but the Achilles' heel of analysis and design methods has been the transition to code. Most of the OO methodologies [4, 5, 6, 7] describe in sufficient detail the steps to be followed during the analysis and design phase, but fail to describe how the analysis and design models of a system shall be converted into implementation code. A big problem in the development of a system through OO methodologies is that even after having created good models, it is difficult for a large fraction of programmers to convert the models into executable code.

There are two major approaches used for OO model-based code generation, namely structural and behavioral. The structural approach is based on using models of object structure (static relationships). It generates code frames (such as class interface specifications) from models of static relationships among objects. Class diagrams concepts can be implemented in a programming language supporting concepts like classes and objects, composition and inheritance. Based on the partial models of object dynamics, developers then explicitly program object behavior and communications in the target language. Many OO CASE tools (ArgoUML [8], Poseidon [9], Metamill [10], etc.) generate limited skeleton code from such models. The main

*Department of Computer Science
Graduate School of Systems and Information Engineering
Tsukuba, Ibaraki, 305-8573, Japan
ianiaz@iplab.cs.tsukuba.ac.jp, jiro@iplab.cs.tsukuba.ac.jp

drawback of this approach is that there is no code generation for object behavior and thus the code generated is not complete.

The behavioral approach extends the techniques used in structural approach. It can generate complete code using additional state machine models and action specifications in a high-level language. With models of both object structure and state machines, this approach enables the tools to generate code for the entire application model.

Our approach for code generation is a behavioral approach. The objective of the current study is to find a method to generate efficient code from the UML class and statechart diagram in an OO programming language. Through mapping between UML and Java, we are able to generate low-level Java code directly from the class and statechart diagram. Code generated from our approach is intended to be complete and covers all the information from the input models. The proposed implementation techniques are valuable in that they raise the level of abstraction and allow for straightforward mapping of UML statecharts to compact and efficient code.

Our code generation approach for implementing statecharts is motivated by [18] and is based on [19], [20] and State pattern [15]. In [18], an implementation model is presented to convert UML statecharts to Java code. The concept of helper object is introduced which handles all state-specific requests forwarded to it by the multi-state domain object. The helper object represents the current state of the domain object. When the state of the of the domain object changes, a new helper object, implementing the behavior specific to the new state, replaces the old one. The proposed model generates code only for the class with which the statechart is attached and other classes of the application model are not considered. The generated code is incomplete and the model is not implemented in any code generating system.

In [19] and [20], we use a different approach. Instead of using helper object, we adapt the idea of State pattern [15] for representing states as objects and provide support for hierarchy, concurrency and the dynamic parts of the statecharts. State pattern puts all the behavior associated with a particular state into one class. The object with state behavior is split into context and a state. The context object contains the common elements of the object's state and delegates events for processing to its current state object. The state object contains state-specific attributes and implementation for state-dependent behavior. The implementation for the history state and fork is not encapsulated rather it is distributed among the context class, state class and the composite state class. In this approach, the code is generated only for the class with which the statechart is associated and the code generation for the other classes in the application model is not considered. The generated code is incomplete.

In this paper, we have used the behavioral approach which is different from the approach of [19] and [20]. This paper focuses on the complete code generation for the entire application model including the class diagram and the statechart diagram. The code generation for some of the statechart features such as history states and fork, is encapsulated in the composite state class to generate complete and more efficient code. The JCode system is developed, which automatically generates the executable Java code using our approach. The results of the experiment, in which the code generated by JCode is compared with the code generated by Rhapsody [14] and OCode [16, 17], is presented in section 7.

The remainder of this paper is organized as follows. In the next section, we present an overview of the features and semantics of the state machine and statechart diagram. Section 3 provides background about various approaches to implement statecharts. Section 4 explains our code generation approach to implement statecharts. In section 5 an air conditioner system example is given to explain our approach. In section 6, we explain our automatic code generating system, JCode. In section 7, code generated by JCode is compared with that of Rhapsody and OCode. Section 8 overviews the related work. Finally, in section 9, we summarize and conclude.

2. State Machines and Statechart Diagrams

A state machine is a graph of states and transitions that describes the response of an object to the receipts of events. State machines are used for specifying the full dynamic behavior of a single class of objects. The diagrammatic presentation of a state machine is a statechart diagram. Figure 1 shows a sample statechart diagram.

Each state models a period of time during the life of an object during which it satisfies certain conditions, performs some action, or waits for some event. A state becomes active when it is entered as a result of some transition and becomes inactive if it is exited as a result of a transition. A transition is a directed relationship between a source state and a target state indicating that an instance in the source state will enter the target state and performs certain actions when a specified event occurs provided that certain specified conditions are satisfied [2]. On such a change of state, the transition is said to "fire". The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. When an event occurs, it may cause the firing of transition that takes the object to a new state. Events are processed one at a time. If an event does not trigger any transition it is discarded.

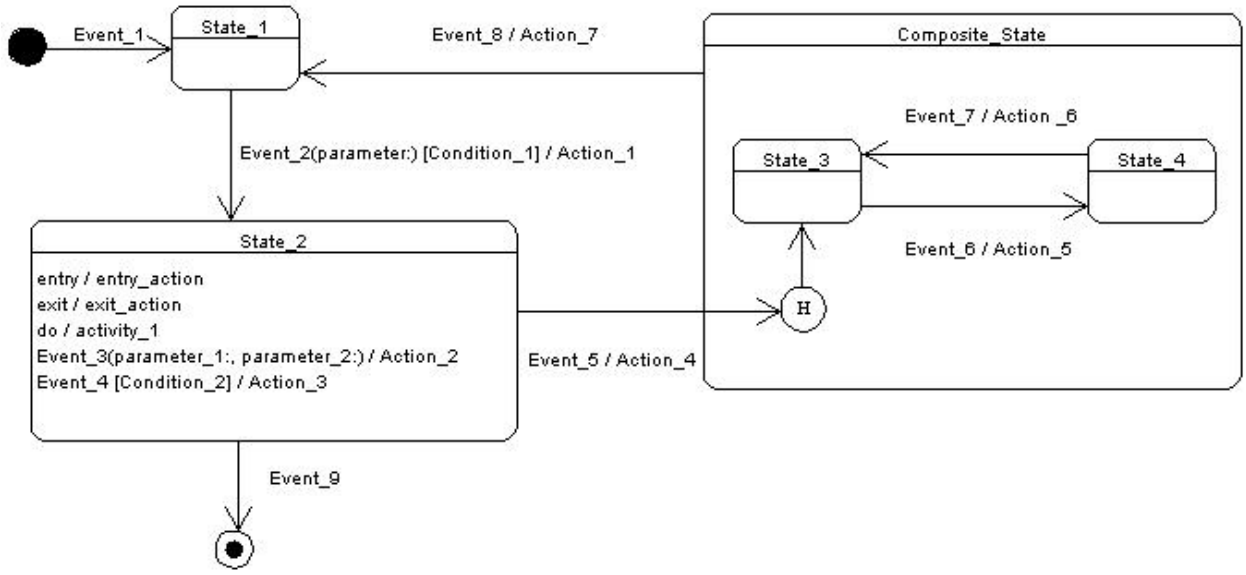


Figure 1. A statechart diagram

A guard condition can be attached to a transition. A guard condition is a Boolean expression that defines conditions under which the transition is able to fire. The guard condition must be true for the transition to be fired. The condition is evaluated at the time the event is dispatched. When a transition fires, an action attached to the transition is executed. The action must be executed entirely before any other actions are considered. A completion transition does not have an explicit trigger event, although it may have a guard defined. It causes an automatic state change immediately after the actions of the source state have been executed.

A state may have, among other features, entry and exit actions, internal activity and internal transitions. Whenever a state is entered, it executes its entry action before any other action is executed. Conversely, whenever a state is exited, it executes its exit action as the final step prior to leaving the state. If defined, the activity (do–activity) associated with the state is forked as a concurrent activity at the instant when the entry action of a state is completed. Upon exit, the activity is terminated before the exit action is executed. A state may also have internal transitions. An internal transition has an event trigger that causes an execution of an action. Firing of an internal transition does not cause a change of state. Therefore, the entry and exit actions of the state are not executed. There are two special states that may be defined for an object's state machine, namely initial state and the final state. The initial state indicates the default starting place for the state machine or substate. An initial state is shown as a small solid circle. The final state indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle.

A statechart may also have composite states. A composite state is a state that contains other states. Any state enclosed within a composite state is called a substate of that composite state. A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates. Given a set of disjoint substates in the context of an enclosing composite state, the object is said to be in the composite state and in only one of those substates at a time. Therefore, sequential substates partition the state space of the composite state into disjoint states. Substates may be nested to any level. A transition may target the composite state or it may target a substate. If its target is the composite state, the nested state machine must include an initial state, to which control passes after entering the composite state and after dispatching its entry action. An initial state is shown as a small solid filled circle. If its target is the nested state, control passes to the nested state after dispatching the entry action (if any) of the composite state and then the entry action (if any) of the substate. A transition leading out of a composite state may have as its source the composite state or a substate. In either case, control first leaves the nested state and its exit action (if any) is executed, then it leaves the composite state and its exit action (if any) is executed. A composite state may contain history state shown as a small circle containing an "H". A history state allows a composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state.

The orthogonal regions represent the parallel composition of sub machines. These substates specify two or more state machines that execute in parallel in the context of the enclosing object. Fork and join pseudostates synchronize transitions entering or leaving orthogonal regions. Whenever there is a transition to a composite state decomposed into concurrent substates, control forks into as many concurrent flows as there are concurrent substates. Similarly, whenever there is a transition from a composite state, control joins back into one. This holds true in all cases. A nested concurrent state machine does not have an initial, final or history state. However, the sequential substates that compose a concurrent state may have these features.

3. Common Approaches to Implement Statecharts

The UML behavior diagrams include many concepts that are not present in most popular programming languages, like C++ or Java, e.g. events, states, history states etc. This means there is not a one-to-one mapping between a statechart and its implementation. A model enables the modeler to work at a higher level of abstraction. The progression from the model to an implemented system is not truly a seamless transition, mainly due to a gap. A model-system gap exists primarily due to the different levels of abstraction. UML is a modeling language, which consists of semantics and graphical notation. For every element of its graphical notation there is a specification that provides a textual statement of syntax and semantics. Implementing the semantics correctly is a challenging task, as the programming languages do not directly support them. The OO programming languages do not deal with abstract behavior.

We observed that states can be represented as scalar variables or they can be represented as objects. Events can be represented as objects or as methods. Ran [11] examined the relation between states and classes. Sane and Campbell [12] proposed that states could be represented as classes and events as operations. Some model elements, like history states, can be implemented in many different ways. This clearly contrasts with class diagrams that often can be implemented directly in a programming language supported concepts like classes and objects, composition and inheritance. We will now discuss some of the common approaches to implement statecharts.

3.1 Switch Statement

The most common technique to implement statechart is to provide a single scalar variable called a state variable and use this as a discriminator in the switch statement inside each event method of the context class [2]. Each case clause in the switch statement can implement the various actions and activities. This technique works well for classical “flat” state machines. The nested states are implemented via many flat states or nested switch statements [13]. The substates are used as a discriminator in the second level of switch statement. All the behavior of the statechart is put in one class. The conditional statements are monolithic and tend to make code less explicit. There is a lot of code duplication and reuse of code is very difficult. Manual coding of entry /exit actions and nested states is, however, cumbersome, mainly because code pertaining to one state becomes distributed and repeated in many places. This makes it difficult to modify and maintain when the topology of state machine changes. It does not provide explicit means for reflecting the transition structure, state hierarchy and entry/exit actions associated to states. Implementing and maintaining the code generated by following this approach is error-prone and labor intensive, but usable in automatic code generators where the code maintenance is substituted by forward engineering. I-Logix’s Rhapsody [14] follows an approach similar to this pattern with major enhancements.

3.2 Helper Object

In [18], the concept of a helper object is introduced, which handles all the state-specific requests forwarded to it by the multi-state object. Multi-state object respond differently to each external message depending upon its current state. The helper object 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 objects responds to the message on behalf of the domain object. When the state of the domain object changes, a new helper object, implementing the behavior specific to the new state, replaces the old one. Helper object puts the behavior associated with a particular state into one object. Events become methods and state hierarchy is implemented by inheritance. Helper object has some similarity with the State design pattern as both represents state as classes and events as methods, but the State pattern neither addresses the state hierarchy nor does it address the concurrency within the statecharts. The implementation of some of the statechart features such as history state and fork is not encapsulated in the composite state rather it is distributed among state objects and the domain object.

3.3 State Design Pattern

By using object orientation, the use of switch statement can be avoided through the use of dynamic binding. State design pattern [15] is the OO replacement of switch statements. Each case becomes a state class and the correct case is selected by looking at the current state. Each state is represented as a separate class. This makes the object’s state as an object in its own right that can vary independently from other objects. States are represented as descendents of a common interface class (each method in this interface corresponds to an event) that declares handler functions for

all events possibly received by the context class. A context class delegates events for processing to the current state object. Transition searching is performed by polymorphism. State transitions are accomplished by changing the current state object. This pattern groups behavior, which is associated to the specific states of the object, into different classes, enabling in this way the separation of concerns in an elegant and efficient way. State pattern facilitates reuse in subclasses because often subclasses that are changed need to be modified. Thus the changes are better encapsulated. The most important weakness of this approach is that it does not provide any means for implementing the dynamic parts of the model. The action chain to be performed on state transitions as defined by UML behavioral model must be coded in the event handler functions. There is no explicit support for hierarchy, history and concurrency.

4. Code Generation Approach

The switch statement solution is not scalable. The code is difficult to read and maintain when the number of states increases. The state pattern does provide a better and scalable solution than switch statement but it still has problems for implementing composite states and other dynamic features of the statechart. The state pattern provides a structural mechanism and the implementation strategy of individual states and sub-statemachines is left open. It provides a general solution and it is not specifically meant for a particular application domain such as implementing the statechart diagram. OCode [16, 17] provided solutions to some of these problems. OCode is a tool for generating code from Object Modeling Technique (OMT) [5] object and dynamic models. OMT state transition diagram is the predecessor of the UML statechart diagram. OCode provides support for implementing composite states but it does not provide support for new features of UML statechart diagram, which are not present in OMT state transition diagram, e.g. history states, fork and join, time events etc.

Our code generation approach is a behavioral approach and it focuses on complete code generation for the entire application model including the class diagram and the statechart diagram. A number of class diagram elements are supported by the Java language, so the translation from class diagram to Java code is relatively straightforward. In contrast, implementing statechart diagram is a challenging task as many concepts are not directly supported by Java.

In our approach, one main application class is generated with a *main()* method that acts as an entry point of the whole application. All the instances of the classes of the class diagram are declared and initialized here. All classes and interfaces within the class diagram are transformed into code. The generated code will contain all the class definitions of name, attributes and methods. Relationships between classes are identified and transformed into code. To implements the association between classes, reference attributes with public visibility are defined in both the classes. If a statechart is attached with a class then the code for implementing the statechart is defined in the corresponding class.

In our approach for implementing statechart diagram, the context class, whose behavior is represented by the statechart, becomes the super context class and defines the interface to clients. An abstract state class is used for defining the interface for encapsulating the behavior associated with a particular state of the context. The abstract state class declares an interface common to all state classes and its purpose is to make all the state classes able to accept every event of the statechart. The interface for internal events and entry /exit actions are also declared in this abstract class. Each state in the statechart diagram becomes a class and is derived from the abstract state class. All the behavior associated with a particular state is put in this state class. Each transition from a state becomes a method in the corresponding state class in order to provide a uniform and convenient way of invoking some services on the context object. If-then statement will be used to check whether the guard condition is satisfied. All the state-specific code resides in one class. The logic that determines the state transitions is partitioned between the state classes. Methods in the state do not need conditional analysis and have no concern for processing in other states. Encapsulating each state transition in a class elevates the idea of an execution state to full object status. Introducing separate objects for different states makes the transitions more explicit. That imposes structure on the code and makes its intent clear. The actions in the transitions of a state machine perform operations on data in the system. We consider actions as messages so each action of the statechart becomes a method in the context class. Internal transitions and entry/exit actions are owned by their containing states so they are implemented as methods in the corresponding state class.

The context object maintains references of all the state objects and they are created once in the constructor of the context object. The context object also holds the reference of the current active state in the *state* object, which is initialized to default state in the constructor of the context. The context has a method for each event of the statechart. Instead of implementing the event method, the context delegates all requests (events) for processing to the current state object. The transition searching is performed using polymorphism. Separating behavior into disparate objects makes sense when the separation takes advantage of polymorphism. Polymorphism allows two objects to be treated identically, even though the objects implement these methods in quite different ways. State transitions are accomplished by changing the current state object. On handling the transitions, the current state object first executes the associated action with the transition followed by the exit action of the current state and then calls the

setState() method of the context to set the new state. In the *setState()* method, the entry action of the new state is also executed. The state object is responsible for specifying the successor state. Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new state subclasses. The abstract state class has an association with the context so it contains a reference to the context object. The state objects inherit this reference to access the methods of the context object.

A composite state is decomposed into two or more concurrent (orthogonal or AND) substates (regions) or into mutually exclusive disjoint (sequential) substates. When a hierarchical composite state containing sequential substates is active, exactly one of its sequential substates is active. Whenever a concurrent composite state becomes active, each one of its concurrent regions also becomes active. The concurrent substates show two or more nested state machines that execute in parallel in the context of the enclosing composite state. This leads us to implement the hierarchical composite state and concurrent composite state by extending the state design pattern with the concept of object composition and delegation. Object composition is defined dynamically at runtime through objects acquiring references to other objects. New functionality is obtained by composing objects to get more complex functionality. Object composition keeps each class encapsulated and there are substantially fewer dependencies. Any object can be replaced at run-time by another as long as it has the same type. Delegation is a way of making object composition powerful for reuse. The main advantage of delegation is that it makes it easy to compose behavior at run-time and to change the way they are composed.

The concurrent composite state becomes the context for all the concurrent regions and maintains references of the current active substates within each concurrent region. Abstract state classes are defined for each concurrent region. Each abstract state class defines an interface for encapsulating the behavior associated with a particular concurrent region of the composite state. The interface for internal events and entry /exit actions are also declared in these abstract classes. Each abstract state class also contains two references, one for the composite state and the other for the super context object to access the action methods of the context object. Each substate in the concurrent region becomes a class and is derived from the abstract state class of the corresponding concurrent region. All the behavior associated with a particular substate is put in this substate class. Each transition from a substate becomes a method in the corresponding substate class. All the substate specific code resides in one class. Internal transitions and entry/exit actions are owned by their containing states so they are implemented as methods in the corresponding substate class.

Whenever a concurrent composite state becomes active, each one of its concurrent regions also becomes active. If the concurrent regions contain history states then the composite state will also maintain references for each history state. These references store the most recent active substate that was active prior to the transition from the composite state. The history reference is updated in the exit action method of the composite object. The current active substates are set in the entry method of the composite object. If the composite state is entered for the first time then default substates are set as active substates. In the other case the history references are used to set the most recent active substate. The implementation of the history state and the fork is encapsulated in the composite state class. The composite state object delegates the incoming requests (events), on which there are transitions within the concurrent region to the corresponding component substate objects. On handling the transitions, the active substate object first executes the associated action with the transition followed by the exit action of the current substate and then calls the *setSub()* method of the composite state object to set the new substate. In the *setSub()* method, the entry action of the new substate is also executed. The substate object is responsible for specifying the successor substate. For transitions that are going out of the composite state or for the internal transitions of the composite state, the composite state object provides the implementation code and does not forward them to the active substate objects. On handling transitions that are going out of the composite state, the composite state object first executes the exit actions of the current active substates, followed by its own exit action and finally the action associated with the transition is executed.

In the case of hierarchical composite state, the composite state will become the context for the nested statechart and will maintain a reference of the current active sequential substate. A composite abstract state class is generated for defining interface for encapsulating the behavior associated with sequential substates. The composite abstract state class contains two references, one for the hierarchical composite state and the other for the super context class. The sequential substates become the concrete substate classes and are derived from the composite abstract state class. The history state implementation is encapsulated in the hierarchical composite class. The composite class keeps the control most of the time and delegates the events to substates for transitions specific to the substates. Our approach translates application model to implementation code and aims not to create excessive information nor result in loss of information during the translation of model to implementation code.

5. Air Conditioner System

We present an example of an Air Conditioner System that will not only clarify the various terms mentioned so far, but will also simplify the coming explanation of our code generation system. Figure 2 shows the static structure of the Air Conditioner System.

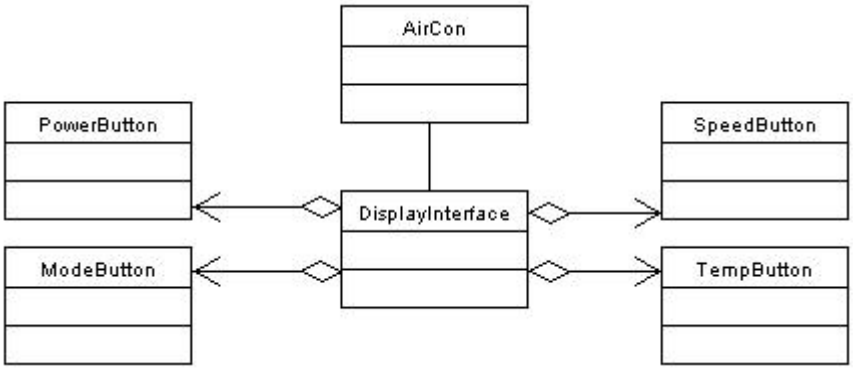


Figure 2. Class diagram of the air conditioner system

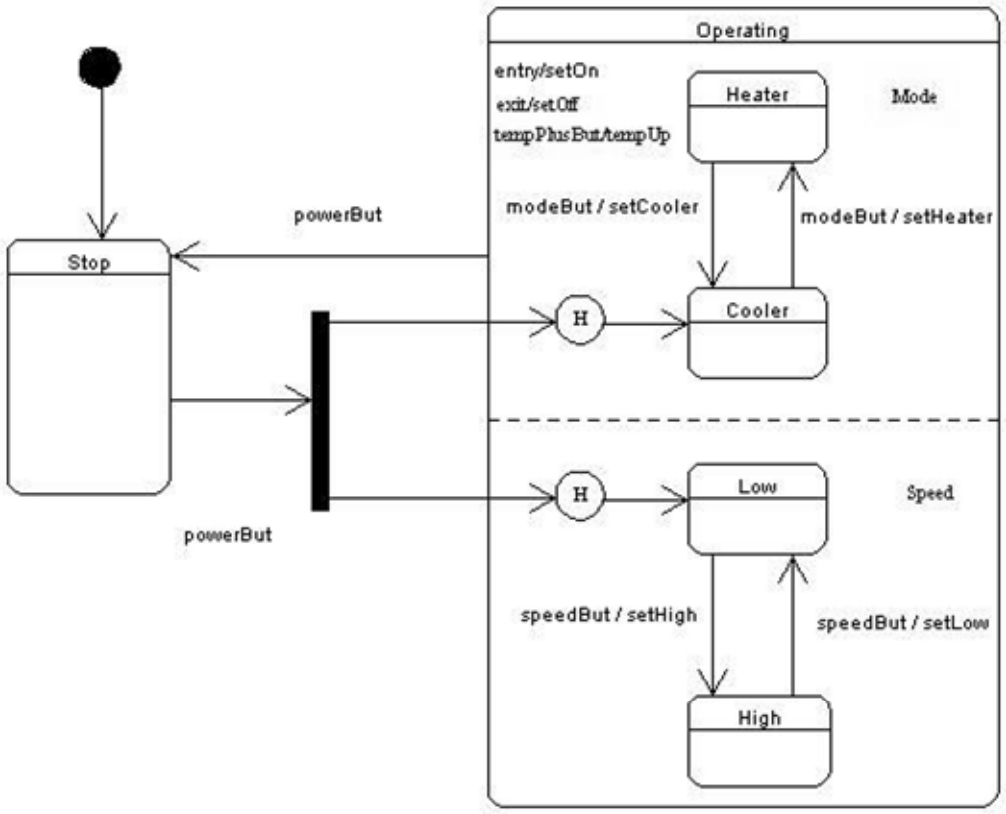


Figure 3. Statechart of AirCon class having concurrent states

The display interface contains four buttons namely, Power, Speed, Mode and Temp and a display area, which shows the current state of the air conditioner. *The DisplayInterface* class maintains one object instance of each of the four buttons *PowerButton*, *SpeedButton*, *ModeButton* and *TempButton* and is associated with the *AirCon* class. The *AirCon* class represents the behavior of the air conditioner. Whenever some button is pressed invoking some service of the air conditioner, the *DisplayInterface* sends the particular message to the *AirCon* class. *DisplayInterface* class acts as a client to the *AirCon* class. The response from the *AirCon* class depends on its current state.

The dynamic behavior of the *AirCon* class is specified in the statechart as shown in Figure 3. It has two top-level states *Stop* and *Operating*. These states are activated alternatively whenever a *powerBut* event occurs. A transition from a solid circle to a state shows that the state is the default state. Initially, the air conditioner is in the default

state *Stop*, where it accepts the *powerBut* event. The air conditioner reacts on such an event by switching from the *Stop* state to the *Operating* state. A state can have entry and exit actions, which are executed when a state is activated or deactivated. When the *Operating* state is activated the *setOn* action is executed and *setOff* action is executed when the *Operating* state is deactivated. A state can also have internal transitions. An internal transition has an event trigger that causes an execution of an action without causing a change in state. While in *Operating* state, if the *tempPlusBut* event occurs then only the *tempUp* action will be executed and the air conditioner will remain in the *Operating* state.

The *Operating* state is a composite state with two concurrent regions *Mode* and *Speed*. These regions become active at the same time whenever the *Operating* state gets activated. Each of the concurrent regions has a number of sequential substates. Only one of the sequential substates becomes active at a given time. While in *Operating* state, on *modeBut* event, the air conditioner switches to the next sequential substate in the *mode* region. Similarly, on *speedBut* event, the air conditioner switches to the next sequential substate in the *speed* region. On *powerBut* event, the air conditioner switches to the *Stop* state. Sending a *powerBut* event will reactivate the air conditioner. When the air conditioner is reactivated, it switches into the history states of the two concurrent regions of the *Operating* state and recalls the last active substates of the two regions. A statechart describes the dynamic aspects of an object whose current behavior depends on its past. A statechart in effect specifies the legal ordering of states an object goes through its lifetime. History state allows a composite state that contains sequential substates to remember the last substates that was active in it prior to the transition from the composite state.

6. JCode

Using our approach, described in section 4, we have developed a system, JCode, which automatically generates Java code from the specifications of the UML class and statechart diagrams. First, the system interprets the model specifications and transforms them into a table and then it generates the actual Java code from the table.

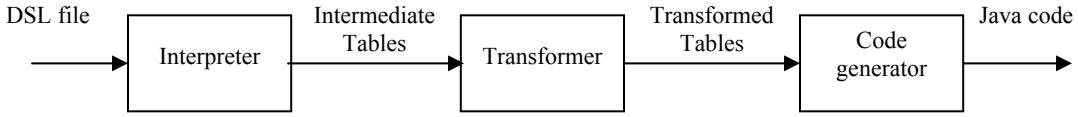


Figure 4. Overall structure of the JCode system

```
OSTD (AirCon)[nodes{n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12},arcs{a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11}];
OSTDN(n1)[loc(25:20),size(20:20),ostdnAttr(name:START)];
OSTDN(n2)[loc(10:140),size(75:125),ostdnAttr(name:Stop)];
OSTDN(n3)[loc(125:160),size(10:100),ostdnAttr(name:FORK)];
OSTDN(n4)[loc(160:10),size(260:400),ostdnAttr(name:Operating,entry/setOn,exit/setOff,event(name:tempPlusBut)/te
mpUp,concurrent{n5,n9})];
OSTDN(n5)[loc(160:20),size(260:180),ostdnAttr(name:Mode,substates{n6,n7,n8})];
OSTDN(n6)[loc(190:150),size(30:25),ostdnAttr(name:HISTORY)];
OSTDN(n7)[loc(260:140),size(70:50),ostdnAttr(name:Cooler)];
…….
OSTDA(a1)[from(n1,side:BOTTOM,off:5),to(n2,side:TOP,off:40)];
OSTDA(a2)[from(n2,side:RIGHT,off:35),to(n3,side:LEFT,off:140),ostdaAttr(name:powerBut)];
……..
```

Figure 5: Statechart specification of the Air Conditioner in DSL format

The input to the system is the model specifications in Design Schema List Language (DSL) [21]. The output from the system is the Java code. DSL is a specification language to represent class and statechart diagram in an understandable text format and to facilitate data exchanges among tools and members of the group. JCode is developed in Java and is basically composed of three modules: *Interpreter, Transformer* and *Code Generator*. Figure 4 shows the overall structure of JCode.

6.1 Interpreter

The interpreter module reads the specifications of the UML statechart diagram, given in DSL format and identifies various states, their substates, history states, transitions and internal events. It makes a nested table to properly record all the information, thus transforming the information from DSL format to a table format. The table is represented by an object structure. Figure 5 shows a part of the DSL representation of the statechart for AirCon class (Figure 3). In DSL, each statement is terminated by a semicolon. The first line declares the nodes and arcs, which compose a diagram. In the diagram, nodes mean states and arcs mean transitions. Each node is described by a separate statement starting with the string "OSTDN", and similarly, each arc is described by a separate statement starting with "OSTDA". Figure 6 shows the table created by the Interpreter module from the statechart diagram of the AirCon class (Figure 3).

6.2 Transformer

This module transforms and organizes the table, created by the interpreter module, in such a way so that code can be easily generated from it. A pseudostate is an abstraction that encompasses different types of transient vertices in the statechart diagram. But DSL, being graphical oriented, treats it as a node like any other node. Transformer module removes the pseudostates (start state, history state, fork and join) from the table and adjusts the information for the affected states and transitions. It sorts the table so that superstates should always come before their substates and puts all the events and actions in the context class. The code generator module needs to know not only the events that are supposed to occur on a state but also the events that may occur on its substates. The transformer module also adds the substates events information to the composite state. Figure 7 shows the state table after transformation performed by the transformer module.

6.3 Code Generator

This module takes information from the table refined by the transformer module and generates the Java language code. It calls various methods, which generates code for the respective classes. Figure 8 shows part of the actual code generated from the air conditioner system of Figure 2. While generating code, the system follows our approach described in previous sections. The detailed rules for code generation are as follow:

1. The main application class *AirConditioner*, is generated. The name of the class is derived from the project name specified in the input DSL file. All the instances of classes of the class diagram are declared and initialized in the constructor of this class. The *AirConditioner* class contains the *main()* method that serves as an entry point. The application object is created and initialized in the *main()* method. The initialization code is also defined here.

State ID	State Name	Substates	Outgoing Transitions				Entry Action	Exit Action	Internal Event
			ID	Event	Action	Next State			
n1	START		a1			n2			
n2	Stop		a2	powerBut		n3			
n3	FORK		a3			n6			
			a4			n10			
n4	Operating	n5,n9	a5	powerBut		n2	setOn	setOff	tempPlusBut/ tempUp()
n5	Mode	n6,n7,n8							
n6	HISTORY		a6			n7			
n7	Cooler		a7	modeBut	setHeater	n8			
n8	Heater		a8	modeBut	setCooler	n7			
n9	Speed	n10,n11,n12							
n10	HISTORY		a9			n11			
n11	Low		a10	speedBut	setHigh	n12			
n12	High		a11	speedBut	setLow	n11			

Figure 6: Table created by the Interpreter module for the air conditioner statechart

State ID	State Name *=default +=history	Substates	Outgoing Transitions				Entry Action	Exit Action	Internal Event
			ID	Event	Action	Next State			
n2	Stop*		a2	PowerBut		n4			
n4	Operating	n5,n9	a5	PowerBut		n2	setOn	setOff	tempPlusBut /tempUp()
n5	Mode+	n7,n8							
n7	Cooler*		a7	ModeBut	setHeater	n8			
n8	Heater		a8	ModeBut	setCooler	n7			
n9	Speed+	n11,n12							
n11	Low*		a10	SpeedBut	setHigh	n12			
n12	High		a11	SpeedBut	setLow	n11			

Figure 7: State table after the Transformer module for the air conditioner statechart

2. One Java class is created for each of the class in the class diagram. The generated code contains all the class definitions of name, attributes and methods. To implements the association between classes, reference attributes with public visibility are generated in both the classes. If a statechart is attached with a class then the code for implementing the statechart is also generated in the corresponding class.
3. The context class *AirCon*, with which the statechart is attached, is generated. It contains an attribute *state* that holds the reference of the current active state of the statechart. All the state objects are also defined here and created once in the constructor of the *AirCon* class. For each event in the statechart diagram, a method is defined that delegates the event to the *state* object. For each action in the statechart diagram, a method is declared in this class. The user enters the body code for the action methods. The default state is also set in the constructor of the *AirCon* class. The *setState()* is also defined, which is used for setting the next state and also calling the entry action of the next state of the statechart
4. To provide a common interface to all state classes, an abstract state class, *AirConState*, is defined. It contains an attribute for the context object and contains empty declarations for all the events in the statechart diagram. Each state class has implementation code for its own events. States in the statechart diagram may have *entry* and/or *exit* actions [2], which are executed whenever the corresponding state is entered or exited. Such actions are implemented as entry and exit methods in the corresponding state classes. *AirConState* provides empty declarations for the entry and exit operations.
5. A class is defined for each state (we call such classes as state classes). The name of the class is derived from the name of the state. All the behavior associated with a state is put in the respective state class. The top-level state classes are derived from *AirConState* class and the substate classes are derived from the corresponding abstract composite state class. If a state has entry/exit actions, methods having the name *entry* and *exit* respectively, are defined in the class. Bodies of these methods contain a method-call to the corresponding entry/exit actions.
6. If a state is a composite state (e.g. *Operating* in Figure 3), the corresponding class contains as many objects as there are concurrent regions in the composite state. For each concurrent region, an attribute with private visibility is defined. If the composite state contains history states then attributes with private visibility are also defined for keeping the reference of the last active substate. The name and type of the attributes are derived from the concurrent region name. The implementation of the history state and fork is encapsulated in the concurrent composite state class. An entry method is defined, which sets the default state (if the composite is entered for the first time) or sets the last active substate (to implement history state). The entry actions of the active substates are also called here. The composite state class is responsible for implementing the fork. Also, an exit method is defined which contains a call to the exit action of the composite state. It also contains the code for storing the active substate in the history state attribute. For each event on the substates, a method is defined that delegates the event processing to the substate and calls the method(s) for that event defined in the class(es) for the substate(s). It also contains methods for setting the next substate and calling the entry action of the next substate.
7. If the state is a concurrent region (e.g. Mode and Speed in Figure 3), the class becomes an abstract class and serves as an interface for its own subclasses. This class is not subclassed from any other class. In addition to the entry and exit operations, it contains empty declarations for operations corresponding to its substates. It also contains two objects, which provide references to the composite state class and to the context class for executing the actions associated with events.

```
class AirCon { // context class
  public AirConState  state; // reference for state
  public Stop stopState;
  public Operating operatingState;
  public Cooler  coolerState;
  public Heater  heaterState;
  public Low   lowState;
  public High  highState;
AirCon() {  //constructor
  stopState = new Stop(this);
  operatingState = new Operating(this);
  coolerState = new Cooler(this,operatingState);
  heaterState = new Heater(this,operatingState);
  lowState = new Low(this,operatingState);
  highState = new High(this,operatingState);
  state = stopState // setting default state          }
public void setState(AirConState st) {
   state = st;
  state.entry();  }
public powerBut() { state.powerBut(); }
public modeBut() {state.modeBut(); }
…….
public void setOff() {……}
public void setCooler() {……}
……}

public AirConState { // abstract state class
  public AirCon ac;  // context reference
  public void entry() {};
  public void exit() {};
  public void powerBut() {};
  public void tempPlusBut() {};
……..
}
```

```
class Operating extends AirConState{ // composite
  private AbsModeState  modeState;
  private AbsModeState  modeHistory;
  private AbsSpeedState  speedState;
  private AbsSpeedState  speedHistory;
  int hist = 0;
public void entry() {
  if (hist > 0) {// last active substate
   modeState = modeHistory;
   speedState = speedHistory; }
else {  // for first time entry
  modeState = ac.coolerState;
  speedState = ac.lowState; }
modeState.entry(); speedState.entry(); ac.setOn(); }
public void exit() {
  ac.setOff; modeHistory = modeState;
  speedHistory = speedState; }
public void modeBut() { modeState.modeBut(); }
public void speedBut() { speedState.speedBut();
public void powerBut() {modeState.exit();
 speedState.exit(); exit(); ac.setState(ac.stopState); }
public void setMode(AbsModeState subMode) {
   modeState = subMode;   modeState.entry();  }
…….}

class AbsModeState {// abstract composite state
  public AirCon m_context;
  public Operating s_context;
/* Empty declarations for entry(), exit() and all
events methods of subclasses of AbsModeState*/ }
class Cooler extends AbsModeState {
 void modeBut() { m_context.setCooler(); exit();
  s_context.setMode(m_context.heaterState); }  }
…….
```

Figure 8: Part of the generated code for the air conditioner system

8. An event on any state becomes a method in the corresponding class. Body code for the method is also completely generated. If the event is an internal event, the body code contains a method-call, which executes the associated action. If the event has a transition, the body code also contains: (i) call to the exit operation of the current state, (ii) method-call for setting the next state and (iii) call to the entry operation of the new state.

7. Comparison with Rhapsody and OCode

Rhapsody [14, 22], which is a successor of STATEMATE [23], is a CASE tool that allows creating UML models for an application and then generates C, C++ or Java code for the application. Rhapsody uses Object eXecution Framework (OXF) [14]. 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.

OCode [16,17] is another tool for code generation from OMT object and dynamic models. OMT state transition diagram is the predecessor of UML statechart diagram. UML statechart diagram contains many features which are not present in OMT state transition diagram e.g. history states, fork and join, time events etc.

7.1 Code Generated by Rhapsody

Rhapsody uses data values to represent states and the operations in the *Reactive* class (inner class within context class) check the data explicitly. Events are represented as classes and are derived from the framework class. The client class creates the event object and calls the *gen* method of the context class. The context class delegates it to *Reactive* class to consume the event. The state transitions are implemented as assignment to some variables and have no explicit

representation. The transition-searching is performed by executing a switch statement in the *Reactive* class. The hierarchical and concurrent substates are handled inside the switch statement. Each action is implemented as a simple statement. The designer user is forced to use target language syntax for defining actions on the transitions. If the action is a method call then no method header is generated, which gives an error while compiling the code.

7.2 Code Generation by OCode

In OCode, states are represented as classes and events and actions as methods. The *Controller* class, which keeps the control of the entire system, contains a helper object to maintain a reference of current state object. The state hierarchy is implemented using the concept of inheritance. The super state of concurrent substates is implemented as a composite class that owns objects of other classes. The composite class maintains objects corresponding to each of the concurrent substates. When the superstate becomes active, the corresponding state class gets instantiated and it instantiates its own substate objects. The state objects are created each time an event is processed. The object creation is an expensive operation. As OMT state transition diagram does not contain history states and fork, we have rewritten the equivalent code for the air conditioner system example in order to have a fair comparison with the code generated by our approach.

7.3 Code Generated by JCode

In the code generated by JCode, the class with which the statechart is attached becomes the context class. The states become classes (inherited from a common interface class). Each action is implemented as a method of the context class and the method header is generated. The user has to write the body code of the action. Hierarchical and concurrent substates are implemented by object composition and delegation. JCode converts each event into a method call and transition searching is performed by polymorphism. The transition code is put in separate methods in the corresponding state classes. All the states and transitions are thus made explicit without using any conditional statement. The context class maintains a reference to the current state object and delegates the event to the current active state. All the states objects are created only once in the constructor of the context class and this leads to faster execution of the event handling mechanism. The hierarchical and concurrent substates are implemented using the concept of object composition. The composite state becomes the context for the nested statechart and contains references for the current active substate. The event is first handled by the composite state. If the target of the event is a substate then it is delegated to the current active substate.

7.4 Comparing the Code Generated by Rhapsody, OCode and JCode

We have used the same air conditioner system (Figure 2) example and compared the code generated by Rhapsody and OCode to that of JCode. Findings of the comparison are as follows:

1. *Code generated by JCode is more compact.* The source code generated by Rhapsody is approximately four times longer than the code generated by JCode, as shown in Table 1. In addition, as the context class and events become subclasses of the OXF framework, the number of classes is much larger than that of JCode. The code generated by JCode is also 10% more compact than that of OCode. The OCode generates the same number of classes.

2. *Our code is more efficient.* To compare the efficiency of the code generated by Rhapsody, OCode and JCode, we performed an experiment in which the same sequence of 4000 requests was sent to the AirCon (context) class that corresponds to the statechart diagram. Out of these 4000 events, 2250 caused transitions while the remaining 1750 events did not cause any transition and were ignored. For each event, the time taken to process the event was calculated. We made all the actions methods empty and concentrated on measuring the time taken while executing transitions, i.e. changing states. To have more accurate results, we repeated the experiment 20 times and calculated the average values. The experiment was performed on Sun SPARC workstation. According to the results of the experiment in Table 2, to process an event that has no transition, our code is 58.50% more efficient than Rhapsody and 15.10 % more efficient than OCode. For events having transitions, our code offers a 71.00% improvement over Rhapsody and 57.60% over OCode. The overall improvement that JCode offers for all type of events is 68.00% over Rhapsody and 49.90% over OCode .

   In Rhapsody, an object instance is created for each event by the client. Various framework classes are involved in the invocation of the actual event processing mechanism and finally the *rootState_dispatchEvent* method of the *Reactive* is called to process the event. This method contains a switch statement that finds if there is any transition on this event from the current state. If there is a transition, the corresponding method is executed, which updates the current state and also calls various methods to perform the entry and exit actions and also executes the associated action. If there is no transition, false value is returned to the calling method. When summed up, all this takes a considerable amount of time.

Table 1. Comparing the compactness of code generated by Rhapsody, OCode and JCode

	Rhapsody	OCode	JCode
Source code: Number of lines	1025	303	273
Source code: Number of bytes	29482	8569	7292
Number of classes	26	10	10

Table 2. Comparing the efficiency of the code generated by Rhapsody and JCode

	Rhapsody(x) (millisecs)	OCode (y) (millisecs)	JCode (z) (millisecs)	Improvement over Rhapsody (x – z)/x*100	Improvement over OCode (y – z)/y*100
Total time for events without transitions (a)	8.80	4.30	3.65		
Average Time per event without transition (a / 1750)	0.00501	0.00246	0.00208	58.50%	15.10%
Total time for events having transitions (b)	28.30	19.35	8.20		
Average Time per event having transition (b / 2250)	0.01257	0.00860	0.00364	71.00%	57.60%
Total time for all events (a + b)	37.10	23.65	11.85		
Average Time per event ((a + b) / 4000)	0.00928	0.00591	0.00296	68.00%	49.90%

In the case of OCode, the state object is created each time the event is handled. The object creation is an expensive operation and takes precious processor time. Even for transitions without events the state object gets instantiated.

In JCode, on the occurrence of an event, the context class delegates it to the current active state. If there is no transition, then only the empty method of the abstract state class is executed, which is a fast operation and nothing more happens. But if the event has a transition, the method in the concrete state class gets executed, which sets the current state reference object of the context class to the new state. All the state objects are created only once in the constructor of the context class. There are no conditional structures in the code. That is why the time taken by our code is markedly short.

3. *Rhapsody code is less understandable.* Rhapsody uses data values to represent states and implements events as classes. The event processing mechanism involves various OXF classes and their methods and is buried in the framework classes. It uses the switch statement for transition searching in each of the methods for a state in the *Reactive* class. This makes the code difficult to understand. Our code converts each event into a method call. The appropriate method is selected on the principle of polymorphism. The transition code is put in separate methods in the corresponding state classes. All the states and transitions are thus made explicit without using any conditional statements. This contributes to making the code more understandable.

8. Related Work

Köhler et al. [24] presented a tool FUJABA [25] for code generation from statecharts. Their approach adapts the idea of generic array based state table [13] but uses an object–oriented implementation of the state table at runtime. Events are implemented as methods. They use objects to represent states of the statechart and attributes to hold the entry and exit methods. These state objects are linked via transition objects that store the triggering event. Additional links and

attributes represent the nesting of complex states, history states etc. A library function is used to interpret the state-table and to react on events. This function is also responsible for issuing appropriate action methods and switching to the resulting states. The state table approach is more complex and causes runtime overhead for table lookups and the interpretative style of event execution. The focus is more on implementing statechart features than efficiency.

Knapp and Merz [26] described a set of tools called Hugo for the code generation of UML statecharts. A generic set of Java classes provides a standard runtime component state for UML statecharts. Every state of a statechart is represented by a separate object, which provides methods for activation, deactivation, initialization and event handling. Events, guards and actions are also implemented as classes. A greedy algorithm performs transition searching. The event traverses the state hierarchy until one or more states are found that consume the event. Simple states determine whether to fire one of their transitions, whereas composite events first let their active substate(s) handle the event before trying to fire their own transitions. In the case of concurrent composite states, the orthogonal regions are traversed in a random permutation. In our approach the composite state first handles the transition and if the target of the transition is a substate then it will delegate it to the current active substate(s). Hugo code generation is interpretative in nature and is not producing the optimized code. The history states are also not implemented.

Gurp and Bosch [27] presented Finite State Machines (FSM) framework, which extends State pattern by modeling all the statechart elements as classes. Similar to the state pattern, there is an FSMContext component that has a reference to the current state and all state-specific data (in a repository). State is represented as a state object rather than a state subclass in the State pattern and is associated with a set of transitions. The FSM object receives the incoming events and responds to them by allocating an FSMContext instance. FSMContext object forwards the request to the current State object. The State maintains a list of transition-event pairs. When an event is received the corresponding transition is located and then executed (triggered). The transition object has a reference to the target state and the corresponding FSMAction object. The Transition object knows which FSMAction to execute and resets the current state on the FSMContext object. A state transition in FSM framework is about twice as expensive as in the State pattern implementation for the simple transitions. Transition searching is done by a look up in a hashtable object. The hashtable object maps event names to transitions. FSM framework does not generate optimized code. History states are also not implemented.

Wasowski [28, 29] presented a hierarchical code generator called SCOPE [30]. SCOPE compiles a sublanguage of statecharts supported by visualSTATE [31] and produces C language code. The visualSTATE statecharts are subset of Harel's statecharts [3] incorporating most of the original statechart language including concurrent states, history, internal transitions and other elements. These statecharts are similar to UML statecharts. SCOPE uses flattening in which hierarchical statecharts are converted into parallel Mealy machines and then code is generated. Abandoning the hierarchy may cause exponential growth of the model, which leads to exponential growth of the program size. The flattening technique is mostly useful only for smaller models.

9. Conclusions

An OO approach for generating Java code from the UML statechart diagrams has been described. States are represented as objects and transitions as operations. All the behavior related with a particular state is put into one object and this localizes the state-specific behavior. Because all state-specific code lives in a state subclass, new states and transitions can be added easily by defining new subclasses. Our approach distributes behavior for different states across several state classes. This increases the number of classes but such distribution is actually good as introducing separate objects for different states makes the transitions more explicit. This makes the components of the statechart diagram explicit and the resulting code easier to understand and maintain. Our approach implements the statechart semantics as faithfully as possible and ensures that the resultant code is still consistent with the UML model.

The proposed approach has been implemented in our system, JCode, which automatically converts the UML statechart specifications into Java code. The comparison with Rhapsody shows that the code generated by our system is 68% more efficient and about four times more compact than that of Rhapsody. Our Code is also 50% more efficient that that of OCode.

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

References

[1] Object Management Group (OMG), Unified Modeling Language (UML) specifications version 1.5, 2003. http://www.omg.org/

[2] G. Booch, J. Rumbaugh, and I. Jacobson, "*The Unified Modeling Language: User Guide*", Massachusetts: Addison-Wesley, 1999.

[3] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, vol. 8, no. 3, pp 231-274, Jun. 1987.

[4] G. Booch, "*Object Oriented Design with Applications*", California: Benjamin/Cummins, 1991.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, "*Object-Oriented Modeling and Design*", New Jersey: Prentice-Hall, 1991.

[6] I. Jacobson, "*Object-Oriented Software Engineering: A Use Case Driven Approach*", Massachusetts: Addison-Wesley, 1992.

[7] International Business Machines (IBM) Corporation, Rational Unified Process, 2003, http://www-306.ibm.com/software/awdtools/rup

[8] Tigris.org, ArgoUML, http://argouml.tigris.org

[9] Gentleware AG, Poseidon for UML, http://www.gentleware.com

[10] Metamill Software, Metamill, http://www.metamill.com

[11] A. S. Ran, "Modeling States as Classes", in *Proc. Technology of Object-Oriented Languages and Systems Conference*, 1994.

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

[13] B. P. Douglass, "*Real Time UML – Developing Efficient Objects for Embedded Systems*", Massachusetts: Addison-Wesley, 1998.

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

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*", Massachusetts: Addison-Wesley, 1995.

[16] 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, pp. 65-77.

[17] 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, pp. 24-34.

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

[19] I. A. Niaz and J. Tanaka, "Code Generation from UML Statecharts", in *Proc. 7th IASTED International Conf. on Software Engineering and Application (SEA 2003)*,.Marina Del Rey, USA, Nov. 2003, pp. 315-321.

[20] I. A. Niaz and J. Tanaka, "Mapping UML Statecharts to Java Code", in *Proc. IASTED International Conf. on Software Engineering (SE 2004)*, Innsbruck, Austria, Feb. 2004, pp. 111-116.

[21] M. Harada, T. Fujisawa, M. Teradaira, K. Yamamoto, and S. Hamada, "Refinement of Dynamic Modeling of Some Automatic Layouting of Object Oriented Design Schema and Reverse Engineering of Design Schema from C++ Program", in *IPSJ Object-Oriented Symposium*, Tokyo, Japan, 1996, pp 111-118.

[22] D. Harel, and E. Grey, "Executable Object Modeling with Statecharts", *Computer*, vol. 30, no. 7, 1997, pp. 31-42.

[23] D. Harel, and A. Namaad, "The STATEMATE Semantics of Statecharts", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, 1996, pp. 293-333.

[24] H. J. Köhler, U. Nickel, J. Niere, and A. Zündorf, "Integrating UML Diagrams for Production Control Systems", in *Proc. 22nd International Conf. on Software Engineering (ICSE 2000),* Limerick, Ireland, 2000, pp. 241-251.

[25] Fujaba Case Tool, http://www.fujaba.de/

[26] A. Knapp and S. Merz, "Model Checking and Code Generation for UML State Machines and Collaborations", in *Proc. 5th Workshop on Tools for System Design and Verification*, Reisenburg, Germany, 2002, pp. 59-64.

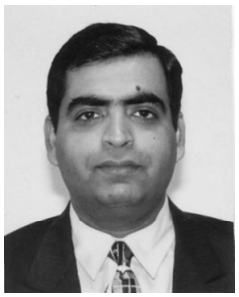
[27] J. V. Gurp and J. Bosch, "On the Implementation of Finite State Machines", in *Proc. IASTED International Conf. on Software Engineering and Applications, (SEA'99)*, Scottsdale, AZ, USA, 1999, pp. 172-178.

[28] A. Wasowski, "On Efficient Program Synthesis from Statecharts", in *Proc. ACM SIGPLAN Conf. of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, USA, June 2003, pp. 163-170.

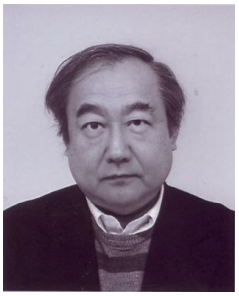
[29] A. Wasowski, "Flattening Statecharts without Explosions", in *Proc. ACM SIGPLAN Conf. of Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington DC, USA, June 2004, pp. 257-266.

[30] SCOPE: A statechart compiler, http://www.mini.pw.edu.pl/~wasowski/scope.

[31] IAR Systems, visualSTATE Case Tool, http://www.iar.com/Products/VS/



Iftikhar Azim Niaz is currently a PhD candidate at the Graduate School of Systems and Information Engineering, University of Tsukuba. His research interests include object-oriented software engineering, design patterns and human computer interaction. He received his MSc from Quaid-i-Azam University in 1994 and MBA from Allama Iqbal University in 1999. He is a member of the ACM, IEEE Computer Society and Computer Society of Pakistan.



Jiro Tanaka is a professor in the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba. His research interests include visual programming, interactive programming, computer-human interaction and software engineering. He is especially interested in the software design methodologies based on object orientation. He received a BSc and a MSc from the University of Tokyo in 1975 and 1977. He received a PhD in computer science from the University of Utah in 1984. He is a member of the ACM and the IEEE Computer Society.