Implementing the Dynamic Behavior Represented as Multiple State Diagrams and Activity Diagrams

Jauhar Ali[1] and Jiro Tanaka[2]
Institute of Information Sciences and Electronics,
University of Tsukuba, Japan

Abstract
A system is introduced which automatically generates implementation code from the object and dynamic models of an application. We found that the behavior of active objects can well be represented by activity diagrams rather than state diagrams. The paper first explains our approach to convert state diagrams as well as activity diagrams into implementation code. The paper then describes our system, dCode, which automatically generates executable Java code from the object diagram, state diagrams and activity diagrams of an application. The paper also presents the results of the experiment in which the code generated by dCode was compared to that of Rhapsody.

Keywords:
Code Generation, OOA/OOD, Dynamic Modeling, CASE

1 Introduction

Object Oriented (OO) software is a collection of interacting objects. Classes of objects have structural properties as well as behavior. There are many OO methodologies [1,2,3,4] that help find the structural properties and behavior of a system by creating its different models. Object Modeling Technique (OMT)[5], which is a popular OO methodology, represents the static aspects of a system by *object model* and its dynamic behavior by *dynamic model*.

OMT and other OO methodologies describe in sufficient detail the steps to be followed during the analysis and design phases but fail to show how the analysis and design models of a system shall be converted into implementation code. For a large fraction of programmers, it is quite difficult to write code from the analysis and design models. This is especially true in the case of the dynamic model. It would be ideal to have CASE tools that can generate or help to generate implementation code from the analysis and design models.

Most of the present CASE tools [6,7,8,9] generate only declarative code like header files from the object model. Generating header files from the object model is straightforward because of its static nature. The dynamic model is considered difficult to implement due to its dynamic nature. A few CASE tools [10,11,12,13,14] can generate code from state transition diagrams. However, most of them do not support state hierarchy and concurrency within state diagrams except Rhapsody [14] which is a tool that generates C++ code from the object and dynamic models.

We have been working on automatic code generation from the dynamic model, which is represented by a set of state transition diagrams. In this context, we already developed a system [15] that automatically converts the dynamic model represented by a simple state diagram into executable Java code [16]. Later, the system was expanded to support concurrent states in the state diagram, thus allowing intra-object concurrency [17]. Both of our previous systems can generate code from the dynamic model that is represented by a single state diagram. However, a real system needs several state diagrams to represent its behavior. In the present study, a system is introduced to automatically generate implementation code from the object and dynamic models having multiple state diagrams. We realized that activity diagrams could well represent the behavior of active objects, which keep their own control. The paper first explains our approach to convert state diagrams as well as activity diagrams into code. While converting the dynamic model into code, we also refer to the object model to generate the initialization code properly. The paper then describes our system, dCode, which automatically generates executable Java code from the object diagram, state transition diagrams and activity diagrams of an application. We also compare the code generated by dCode to that of Rhapsody.

2 The Elevator Example

We illustrate our approach using an application that simulates a system controlling three elevators and six floors. The problem is a simplified version of the lift problem presented at the Forth International Workshop on Software Specification and Design [18]. The system has the following constraints.

1. Each elevator has a set of buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is cancelled when the elevator visits the corresponding floor.

[1] Current affiliation is Software Research Associates (SRA) Inc., 3-12 Yotsuya, Shinjuku-ku, Tokyo 160-0004 Japan
E-mail: jauhar@sra.co.jp
[2] Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
E-mail: jiro@computer.org

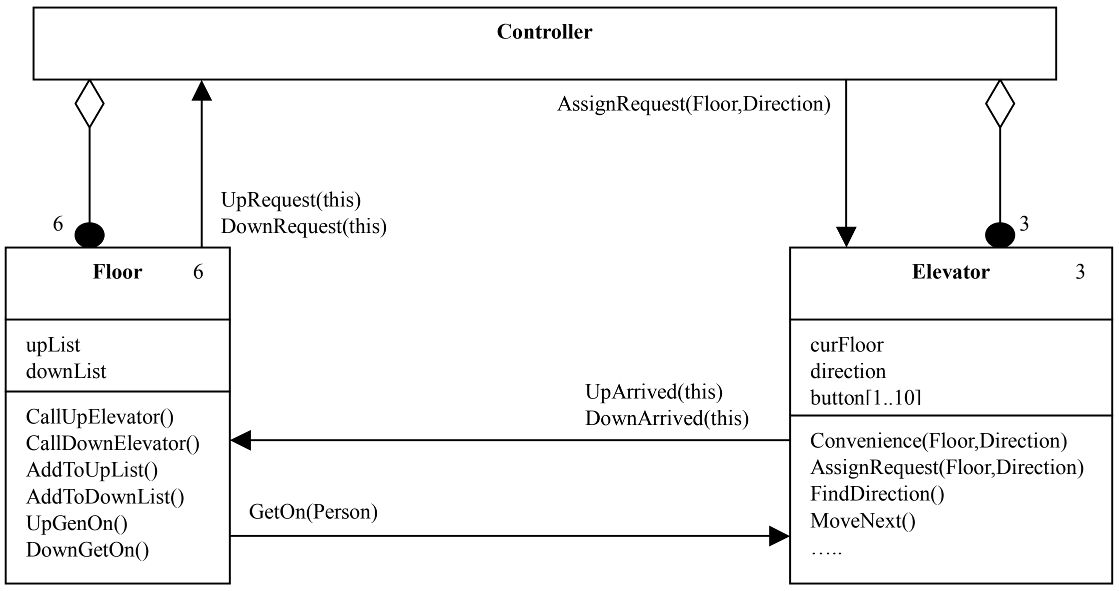


Figure 1: Object model for the elevator simulation system

2. Each floor has two buttons (the ground and the top floors have only one button each), one to request an up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is cancelled when an elevator visits the floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floors' request buttons are pressed, only one should be cancelled.
3. When an elevator has no requests to service, it should remain at its final destination with its doors closed and await further requests.
4. All requests for elevators from floors must be serviced eventually, with all floors given equal priority.
5. All requests for floors within elevators must be serviced eventually, with floors being serviced sequentially in the direction of travel.

3 Designing the Elevator System

To design the elevator simulation system, we believe that there must be a `Controller` class that keeps control of the overall system and at least two more classes: `Elevator` and `Floor`. Figure 1 shows the object diagram. We used the OMT notation except that the arrow headed lines between classes show possible messages between them.

When a button is pressed on a floor, the Floor object sends a request to the Controller for an elevator in the desired direction. The Controller sends a message to each of the Elevator objects asking its convenience for servicing the request. In response, each of the Elevator objects gives an integer value representing its convenience for the request. The elevator which returns the highest value is the most convenient, and the Controller assigns the request to it. Each of the elevators runs in parallel and checks whether it has any outstanding requests. If there is a request, the elevator visits the desired floor and sends an arrival message to the Floor object. The floor gets all the persons that were waiting on the elevator. Each person, while getting on, presses the destination button inside the elevator. The elevator then moves to the required floors in sequential order. When an elevator stops at a floor, it resets the destination button for that floor and gets all the persons off whose destination was that floor.

In a real elevator system, persons come in randomly at various floors and press the buttons in the desired directions. However, to make the system a bit interactive, we made the user (operator) of the system to press the floor buttons by clicking at them with mouse, meaning that a person has come at the floor and has pressed a button in the desired direction. The Floor object then increments the number of persons waiting at the floor. The destination floor for a person is randomly determined. If there is no person waiting at the floor when a floor button is pressed, the Floor object also sends out a `callElevator` message to the Controller.

4 The Controller Class

`Controller` is a special class that keeps the main flow of control of a system [15,17,19,20] and represents the system as a whole. One of the main responsibilities of the Controller is to initialize the system and create permanent objects of the system. Objects are called *permanent* if they exist throughout the system is running. *Temporary* objects are created for a short time while the system is running.

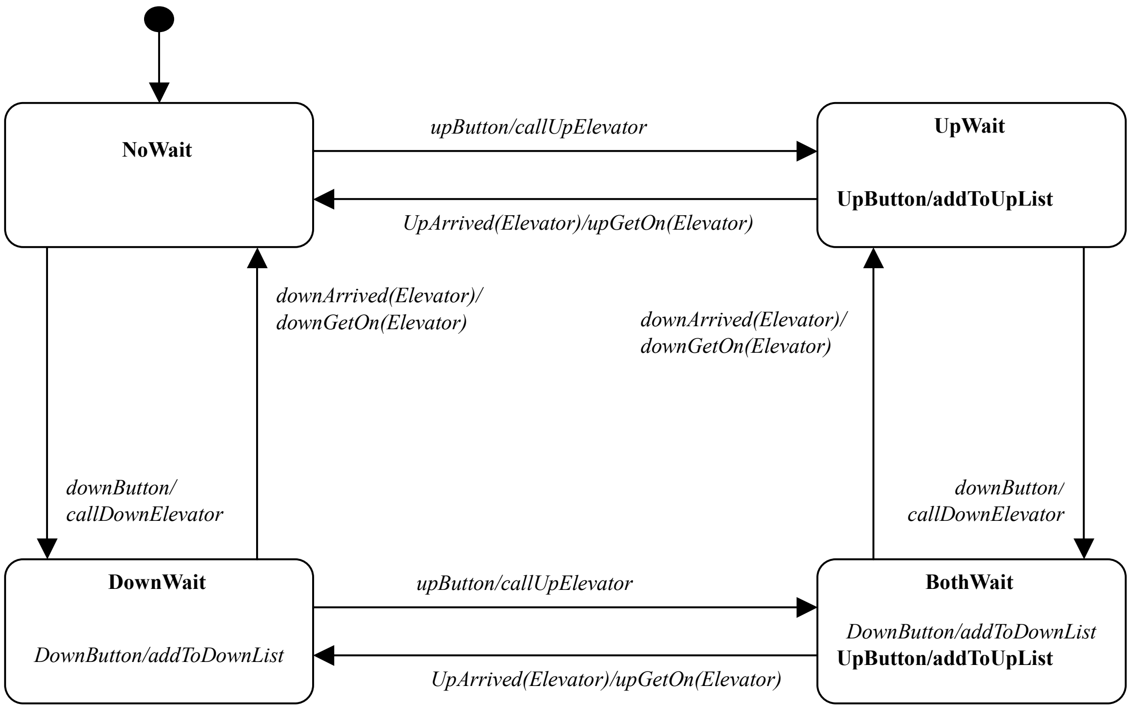


Figure 2: State diagram for `Floor` class

After the initialization of the system, control mostly resides in the Graphical User Interface (GUI) component of the system. Controller receives messages from the GUI when a user interacts with the system. In response to these messages (events), Controller possibly changes the state of the system and sends messages to other objects in the system. Typically, there is only one controller in a system, so there is only one object instance of this class.

The `Controller` class in the elevator example is a *uni-state* class and does not need a state diagram. It initializes the system by creating permanent objects (6 instances of `Floor` and 3 instances of `Elevator`). After initialization, control of the system resides in the click-able buttons that represent the up and down buttons at each floor. While the system is running, the `Controller` always behaves in the same way whenever it receives incoming messages from the Floor objects. So we do not need to do anything special about implementing the dynamic behavior of the Controller.

5 State Diagrams

Objects provide a number of services, which are accessed or get executed by sending a message to them. Objects often have different states and the availability of services provided by the objects depends upon the state they are currently in. Such objects can be named as *multi-state* objects. A state diagram usually represents the behavior of a multi-state object. Unlike the `Controller` class, the `Floor` class in the elevator example, is a multi-state class. To represent the behavior of a multi-state class, we use Harel's statecharts [21,22,23], which can contain OR-type or AND-type state hierarchy. Figure 2 shows the state diagram for the `Floor` class.

5.1 Converting a State Diagram into Code

To implement a state diagram, an object-oriented approach is used where each state becomes a class and each transition becomes an operation in that class. OR-type substates of a superstate become subclasses of the class that corresponds to the superstate. Substates are called OR-type if only one of the substates can be active at a given time when the superstate is active. All the state classes are subclassed from an abstract class that serves as a common interface for the state classes. We named the common interface class for the state classes of `Floor` as `FloorState`. This approach is adopted due to the resemblance of the state hierarchy in a state diagram and the class hierarchy in an OO program. The substates inherit transitions of the superstate just like subclasses inherit the methods of the parent class.

As can be seen in Figure 3, all the actions in the state diagram become methods in the corresponding domain class. For example, the `callUpElevator` and `upGetOn` actions become methods in Floor class. The Floor class maintains an attribute `state` of `FloorState` type. The value of this attribute shows the current state of the Floor

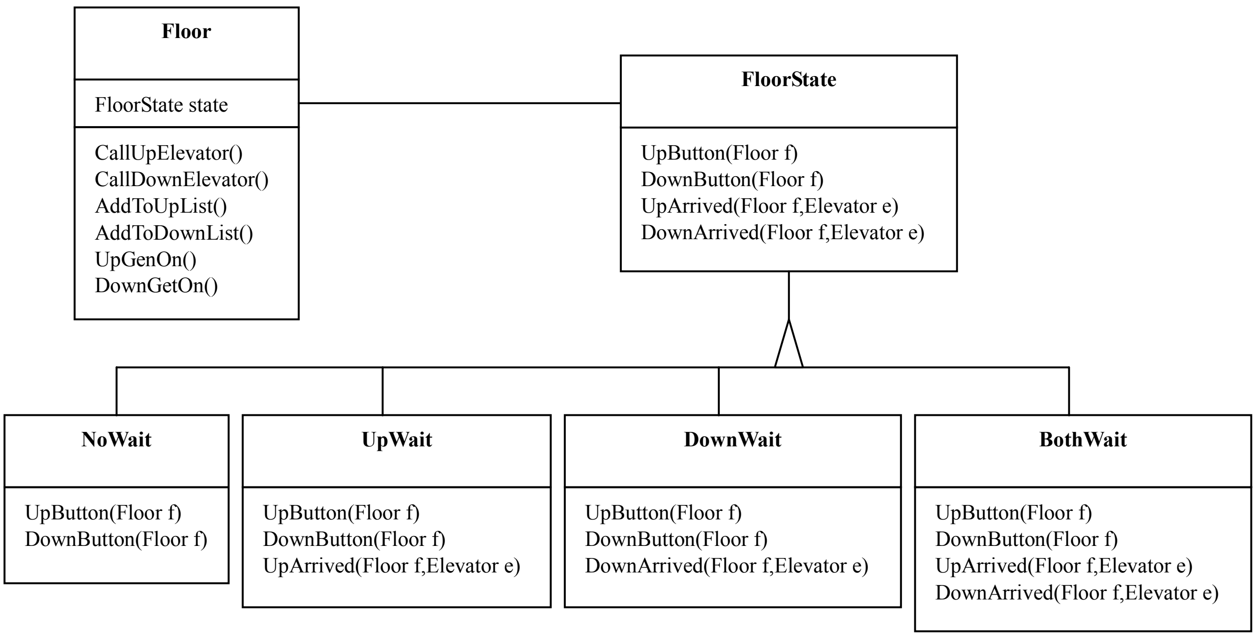


Figure 3: Class structure of the `Floor` class and its companion classes

object. Objects that communicate with the Floor object do not need to know about state classes (subclasses of `FloorState`). They just send requests to the Floor, which become events for it. The Floor object delegates all the requests (events) to its state object. If there is a transition on the event, the corresponding operation in one of the state classes will get executed and the state of the Floor will change.

As there are many Floor objects, the methods of the subclasses of the `FloorState` class, which correspond to state transitions, should know which instance of the `Floor` class is going to change state while the methods are executed. Due to this reason, the Floor object passes itself to the state object as a parameter, whenever it delegates outside requests (events) to the state object. Part of the implementation code for the `Floor` class and its corresponding state classes is shown below.

```
class Floor {
   //default state
   FloorState state = new NoWait();

   //delegates all events to state
   //and passes itself as parameter
   upButton(){
      state.upButton(this);
   }
   upArrived(Elevator e){
      state.upArrived(this, e);
   }
   .....


   //actions in the state diagram
   //become methods here
   callUpElevator(){.....}
   addToUpList(){.....}
   .....
}
class FloorState {
   //contains empty declarations for
   //all methods in the subclasses
}
class NoWait extends FloorState {
   //each transition becomes a method
   upButton(Floor f){
      // call action
      f.callUpElevator();
      //change state
      f.state = new UpWait();
   }
   downButton(Floor f){
      f.callDownElevator();
      f.state = new DownWait();
   }
}
class UpWait extends FloorState {
   upButton(Floor f) {
      f.addToUpList();
   }
   downButton(Floor f){
      f.callDownElevator();
      f.state = new BothWait();
   }
   upArrived(Floor f, Elevator e){
      f.upGetOn(e);
      f.state = new NoWait();
   }
}
.....
```

5.2 Optimizing the Code

As can be seen in the code above, when an event occurs, a method in the abstract class (`FloorState`) is called which is a fast operation. When there is a transition, however, a method in one of the subclasses of the `FloorState` class for that operation is executed. The method dynamically creates a new instance representing the new state. The dynamic creation of objects makes the code a bit less efficient.

Optimization of the code can be achieved by creating an instance of each of the subclasses of the `FloorState` class beforehand and then assigning one of these instances to the state object while a transition is executed. As the subclasses of `FloorState` only contain operations and do not have any data, their instances can be shared among different `Floor` objects. Therefore, we make these instances as class members (`static`) in the `Floor` class. Also, since these instances are only meant to be assigned to the state object and should not be changed, we declare them as constants (`final`). Following is part of the optimized code for the `Floor` class and its associated state classes.

```
class Floor {
   // creat state objects only once
   final static FloorNoWait NO_WAIT
         = new FloorNoWait();
   final static FloorUpWait UP_WAIT
         = new FloorUpWait();
   final static FloorDownWait DOWN_WAIT
         = new FloorDownWait();
   final static FloorBothWait BOTH_WAIT
         = new FloorBothWait();

   //default state
   FloorState state = NO_WAIT;
   .....
}
class FloorNoWait extends FloorState {
   upButton(Floor f){
      f.callUpElevator();
      f.state = UP_WAIT;
   }
   downButton(Floor f){
      f.callDownElevator();
      f.state = DOWN_WAIT;
   }
}
```

5.3 Concurrency within State Diagrams

State diagrams can have concurrent states (AND-states). AND-states become active simultaneously whenever their superstate becomes active. As already described, in our approach an active state is represented by an object instance. For concurrent states, we need a mechanism that guarantees the creation of as many objects as the number of the concurrent substates whenever their superstate becomes active. That is why, we represent the superstate of AND-substates as a composite class that owns objects of other classes.

The composite class has references to the classes corresponding to the AND-substates. When the superstate of AND-substates becomes active, the corresponding composite class gets instantiated. The composite object then instantiates all the classes for which it has references. The instantiation of the composite class thus guarantees the instantiation of the classes that correspond to the AND-substates. This behaves like activating many states simultaneously. Similarly, when the composite object is deleted, all the objects it owns are also deleted. This behaves like leaving all the AND-substates at once when their superstate becomes inactive.

6 Activity Diagrams

State diagrams work well for representing the behavior of *passive objects*, which do not usually keep control. They get control only when some other object sends a message to them causing the execution of one of their methods. After having completed the execution, control is transferred back to the object that had sent the message.

In real systems we sometimes encounter *active objects*, which keep their own control. They mostly perform their operations in a continuous loop. During the execution of the methods, they can send messages to execute methods in other objects and cause a temporary transfer of control to those objects. Control is transferred back to the active objects as soon as the execution of the methods in other objects is finished. For example, Elevator is an active class and does not wait for incoming messages from other objects. Instead, it executes a continuous loop and in each iteration, it checks whether there is any outstanding request that should be serviced.

We observed that state diagrams could not represent well the behavior of active objects, because the transitions in state diagrams are mostly triggered on the occurrence of some external events. We represent the behavior of active objects by an activity diagram. An activity diagram is like a state diagram except that the transitions are not triggered by external events [24]. Each node in the activity diagram shows an activity or possibly a condition, whereas in the state diagram it shows a state. As soon as the activity is performed, the transition is triggered and a new activity starts execution. In an activity diagram, the object itself determines when to execute a transition. It does not have to wait for other objects to send it messages, which become events to trigger the transitions. Figure 4 shows the activity diagram for the `Elevator`.

In the proposed approach, an active class is implemented as a Java thread. Therefore, each Elevator object has its own control. The continuous loop, which can be seen in the activity diagram, is placed inside the `run()` method. Each activity becomes a method in the class. If a node represents a condition, e.g., `MoveNeeded`?, it also becomes a method but returns a `boolean` value. In the loop, each method is called in the sequence in which it appears in the activity diagram. The method that represents a condition is called from inside an `if`-statement. Sub-activities, e.g., `NotifyArrival` and `OpenDoor` inside the `Stop` activity, become separate methods as well, and they are called from the method that corresponds to the super-activity (in this case the `Stop` activity). Following is the implementation code generated from the activity diagram of the Elevator.

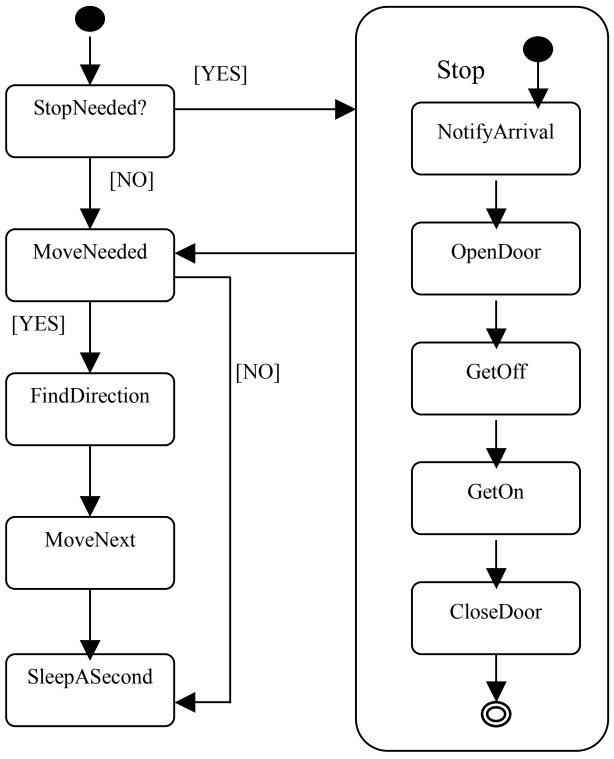


Figure 4: Activity diagram for `Elevator` class

```
class Elevator implements Runnable {
   .....
   public void run() {
      for (;;) {
         if (stopNeeded()) stop();
         if (moveNeeded()){
            findDirection();
            moveNext();
         }
         sleepASecond();
      }
   }
   void stop() {
      notifyArrival();
      openDoor();
      getOff();
      getOn();
      closeDoor();
   }
   .....
}
```

7 Classes having both State and Activity Diagrams

Classes of objects may have more than one aspect, which should be implemented separately. An active object, whose behavior is represented by an activity diagram, can also be a multi-state object and, therefore, need a state diagram showing its multi-state behavior. The `Elevator` class in the example is an active but uni-state class. (Here we are using the term multi-state in its strict meaning that suggests state-transitions triggered by external events, which are asynchronous, rather than internal signals, which merely show the completion of some actions.) Whenever the Controller object sends a message to the Elevator object asking its convenience, the Elevator object returns its convenience as an integer value. Similarly, each time the Controller object assigns an elevator to some floor, the Elevator object records the request as an outstanding request. This uni-state behavior of the `Elevator` class is not shown in the activity diagram (Figure 4). If the Elevator had also been a multi-state class, we would have had to draw a state diagram in addition to the activity diagram, and to implement it in a way similar to the `Floor` class.

Suppose the elevator system has a Halt and a Restart buttons for each elevator, which are used by a maintenance operator to do his maintenance job. The elevator now has two states: Normal (default) and Halted. In the Normal state the elevator acts just like before. However, when the Halt button is pressed, the `halt` action is executed which makes the elevator stop at the current floor and suspends the thread. In the Halted state, the elevator returns INCONVENIENT whenever its convenience is asked by the Controller. The Controller will not be able to assign a request to an Elevator in the Halted state. When the Restart button is pressed, the elevator goes back to the Normal state. This multi-state as well as active behavior of the `Elevator` class can be implemented as follows.

```
class Elevator implements Runnable {
   final static INCONVENIENT = -99999;
   ElevatorState state =
      new ElevatorNormal();
   //delegating state-specific requests
   //to state object
   public int convenience(){
      state.convenience(this);
   }
   public void haltBut() {
      state.haltBut(this);
   }
   public void restartBut() {
      state.restartBut(this);
   }
   // state diagram actions
   public void halt(){
      stop();
      suspendThread();
   }
   public void suspendThread(){
      //some code
   }
   public void resume(){
      //some code
    }
    // activity diagram code
    public void run(){
      // as before
    }
    .....
  }
  class ElevatorState {
    //contains empty declarations for
    //all methods in the subclasses
  }
  class ElevatorNormal extends
    ElevatorState {
    public int convenience(Elevator e){
      // as before
    }
    public void haltBut(Elevator e){
      e.halt();
      e.state = new ElevatorHalted();
    }
  }
  class ElevatorHalted extends
    ElevatorState {
    public int convenience(Elevator e){
      return e.INCONVENIENT;
    }
    public void restartBut(Elevator e){
      e.resume();
      e.state = new ElevatorNormal();
    }
  }
```

8 The Code Generating System

The proposed approach has been implemented in our system, dCode, which automatically generates executable Java code from the specifications of the object and dynamic models of a system. dCode takes as input specifications of the object diagram, state diagrams and activity diagrams in Design Schema List (DSL) language [25]. DSL is a specification language, which was developed by other members of our research group to easily represent OMT diagrams in an understandable text format, and to facilitate data exchanges among tools and members of the group. Nakashima et al. [26] have developed a tool through which OMT diagrams can be drawn interactively and then translated automatically into DSL. The output from dCode is Java code. The system generates implementation code for different classes in the following way.

8.1 Generating Code for Domain Classes

Classes that appear in the object diagram are called domain classes. Declaration code for the domain classes, which contains attributes and methods, is generated from the information of the object diagram. Detail implementation code for each class is generated depending on the class type and whether it has any associated state diagram and/or activity diagram, as explained below.

8.1.1 Controller

In any application, there is typically one class that plays the role of a controller. The Controller initializes the system and all permanent objects in the system. If the permanent objects are active, new threads for them are also created. The `main()` method is defined in the Controller which instantiates an instance of the Controller. dCode generates the following code for the `Controller` class of the elevator application.

```
class Controller {
  public Elevator[] elevatorList =
    new Elevator[3];
  public Floor[] floorList =
    new Floor[6];
  public Controller() { // constructor
   for (int i=0;i<floorList.length;
        i++)
     floorList[i] = new Floor(i,this);
  for (int i=0;i<elevatorList.length;
         i++) {
  elevatorList[i]=
         new Elevator(i,this);
  new Thread(elevatorList[i]).start();
  }
  }
  public static void
     main(String args[]){
    Controller c = new Controller();
   .....
  }
}
```

8.1.2 Classes having Activity Diagrams

If a domain class has an activity diagram, the class implements the Runnable interface, so that separate threads can be started for its objects while instantiating them. `run()` method is defined in the class, as already explained in Section 6. For each activity in the activity diagram, a method is declared in the class. The user enters body code for these methods.

8.1.3 Classes having State Diagrams

If a domain class has a state diagram, an attribute state is defined that represents the current state of objects of the class. For each event in the state diagram, a method is defined that delegates the event to the state object. For each action in the state diagram, a method is declared. The user enters body code for the action methods.

8.2 Generating Code for State Classes

If a domain class has a state diagram, additional classes are created that implement the state-specific behavior of the class. We call these extra classes as state classes. State classes are generated as follows.

1. To provide a common interface to all state classes, an abstract class is defined. The name of the class is obtained by suffixing “State” to the name of the corresponding domain class. It contains empty declarations of operations for all events in the state diagram. Each state class has implementation code for its own events (operations).
2. A class is defined for each state. The name of the class is

derived from the name of the state. If the state is a substate of another state then it can make an OR-type or AND-type state hierarchy.

a) **OR-Type State Hierarchy:** Classes corresponding to the substates become subclasses of the class that corresponds to the superstate. Transitions from the superstate are implemented as methods in the superclass and are derived in the subclasses. Transitions from the substates are implemented as methods in the subclasses.

b) **AND-Type State Hierarchy:** The class corresponding to the superstate of AND-states becomes a composite class that contains as many objects as the substates. For each substate, an attribute is defined. The name and type of the attribute are derived from the name of the substate. For each event on the substates, a method is defined that calls the method(s) for that event defined in the class(es) for the substate(s). The class that corresponds to an AND-state becomes an abstract class and serves as an interface for its own subclasses.

3. An event on any state becomes a method in the corresponding class. Body code for the method is also completely generated, which contains a call to the action of the transition and code for adjusting the new state.

9 Comparison with Rhapsody

Rhapsody [27,14], which is a successor of O-Mate [28], is a tool that allows creating object diagram, state transition diagrams and message sequence charts for an application and then generates C++ code for the application. Because Rhapsody is the only tool of its kind (we mention some other tools in the next section), we compare dCode to Rhapsody.

Rhapsody does not consider activity diagrams to represent behavior of active objects. It only uses state transition diagrams to represent the behavior of classes of objects, so we can compare only the code generated from state diagrams. The details of converting a state diagram into code are not fully given in the papers [27,28], where the tool is reported. However, the mechanics can be understood by looking at the code generated by Rhapsody. Like dCode, Rhapsody also treats states as separate objects. But the way it handles events and state hierarchy is quite different to that of our approach.

9.1 Code Generated by Rhapsody

Rhapsody represents all states and events as classes. First, four classes: `AndState`, `ComponentState`, `OrState` and `LeafState` are derived from an abstract class `State`. The four classes implement respectively the general behavior of four types of states: superstate of AND-states, AND-state, superstate of OR-states and leaf state. Each state of the state diagram become a class and is subclassed from one of the above four classes depending on the state type. The domain class that corresponds to the state diagram maintains one instance each of all the state classes. Each state object has a pointer to its superstate object, and a method `boolean in()` that returns `true` if the state is active. Similarly, there is a general `OMEvent` class from which all the event classes are derived. For each transition, there is a method defined in the domain class. As shown in Figure 5, when an object of the domain class receives an outside request (event), it creates an event object (steps 1 and 2). Then it calls the `takeEvent (EventId)` method of the active state object and passes the event as an argument (steps 3 and 4). This method contains a *switch* statement that searches out a transition on this event from the current state (step 5). If there is a transition, the corresponding method in the domain class is made executed (step 6a), which updates the current state of the domain class (step 7). If there is no transition, the event is sent to the object representing the parent state (step 6b). All this when sum up takes a considerable amount of time.

9.2 Code Generated by dCode

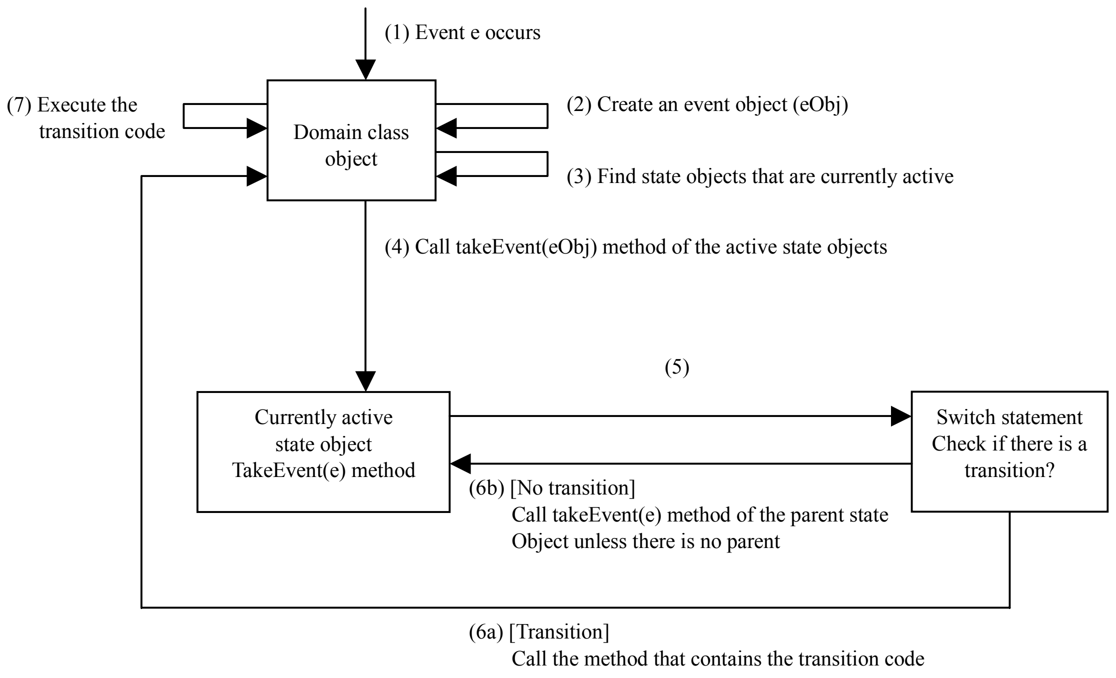
In the code generated by dCode, states become classes (inherited from a common interface class) but events become methods. Because state hierarchy is implemented by the inheritance mechanism, state objects do not need to have pointers to their superstate objects. In each state class, methods are defined that correspond to the transitions going out of the state.

When an event occurs on which there is a transition, the corresponding method in the current state object is executed. If there is no transition on an event, there will be no method in the current state object, and as a result only the empty method in the abstract interface class will be executed. This is a fast operation. That is why, the time taken to process an event without transition is markedly short. In the case of a transition, the time is longer but as the code does not contain any conditional statement, the time is still shorter than that of Rhapsody's code.

9.3 Comparing the Code Generated by Rhapsody and dCode

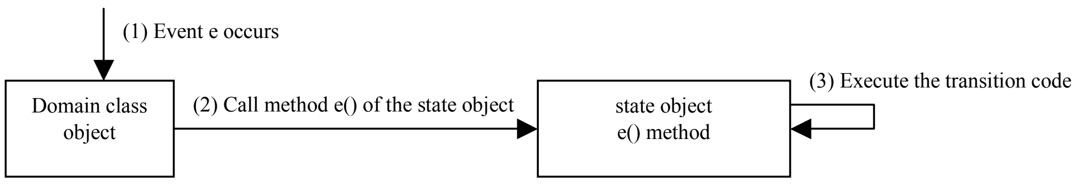
We used a simple example having one state diagram and compared the code generated by Rhapsody to that of dCode. To have a fair comparison, we rewrote the code generated by Rhapsody in Java, because dCode generates Java code whereas Rhapsody generates C++ code. Findings of the comparison are as follow;

1. Code generated by dCode is more compact. The original C++ code generated by Rhapsody was too much long. After rewriting it in Java, the source code becomes shorter but is still approximately five times longer than the code generated by dCode, as shown in Table 1. In addition, as all states and events become subclasses of the various classes explained above, the number of classes is much more than that of our code.
2. Our code is more efficient than Rhapsody's code. To compare the efficiency of the code generated by dCode and Rhapsody, we performed an experiment in which the same sequence of 1000 requests was sent to the class that



(1) Event e occurs
(7) Execute the transition code
Domain class object
(2) Create an event object (eObj)
(3) Find state objects that are currently active
(4) Call takeEvent(eObj) method of the active state objects
(5)
Currently active state object TakeEvent(e) method
Switch statement Check if there is a transition?
(6b) [No transition] Call takeEvent(e) method of the parent state Object unless there is no parent
(6a) [Transition] Call the method that contains the transition code

(a) Execution sequence of the code generated by Rhapsody



(1) Event e occurs
Domain class object
(2) Call method e() of the state object
state object e() method
(3) Execute the transition code

(b) Execution sequence of the code generated by dCode

Figure 5: Conceptual view of the executing code generated by Rhapsody and dCode

corresponds to the state diagram. Out of these 1000 events, 444 caused transitions while the remaining 556 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 action 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 Station 10. According to the results of the experiment in Table 2, to process an event that has no transition, our code is 57.50% more efficient than Rhapsody's code. For events having transitions, our code offers a 20.80% improvement over Rhapsody's code. The overall improvement that dCode offers for all types of events is 38.00%.

	Rhapsody	dCode
Source code: No. of lines	1031	231
Source code: No. of bytes	19891	5410
No. of classes	26	14

Table 1: Comparing the compactness of the code generated by Rhapsody and dCode

3. Rhapsody code is difficult to understand. As explained above, though Rhapsody implements state-specific behavior in separate classes, it puts the transition-selection code in the `switch` statement inside the `takeEvent(EventId)` method of the state classes. Actual transitions are implemented as methods in the corresponding domain class, which are called when the current state object succeeds in finding a transition on

an event. This makes the code difficult to understand. Our code converts each event into an operation call. The appropriate method is selected on the principle of polymorphism. The transition code is put in separate methods in the corresponding state classes. All the states and transitions are thus explicit without using any conditional statements. This contributes to making the code more understandable.

	Rhapsody (*x*) (millisecs)	DCode (*y*) (millisecs)	Improvement *(x-y)/x*100*
Total time for Events without Transitions (*a*)	127.800	54.3000	
Average time per Event without Transition (*a/556*)	0.2299	.0977	57.50%
Total time for Events having Transitions (*b*)	144.7500	114.6500	
Average time per Event having Transition (*b/444*)	0.3260	0.2582	20.80%
Total time for all events (*a+b*)	272.5500	168.9500	
Average time Per event (*(a+b)/1000*)	0.2726	0.1690	38.00%

Table 2: Comparing the efficiency of the code generated by Rhapsody and dCode

10 Related Work

The most related work is that of Harel and Gery [27,28] whose tool, Rhapsody [14], generates C++ code from the object and dynamic models. As shown in the previous section, our code is more compact, efficient and simple than that of Rhapsody.

In addition to Rhapsody, there are other commercially available CASE tools that support graphical editors to draw various OMT diagrams and then generate some of the implementation code from them. The major one is Rational Rose [6], which provides interactive graphical editors to make various UML [24] and OMT diagrams. Because Rational Rose is basically a modeling and documentation tool, it generates only header files from the object model and does not generate any code from the state diagrams. Object Oriented Designer [9], Object Domain [7] and MacA&D [8] are other tools that also generate only header files from the object model. Some of the tools, such as StateMaker [10], ROOM [11], Graphical Designer [12] and StP [15] , can generate code from the state diagrams too, but they do not usually support state hierarchy and concurrent states in the state diagrams.

Our mechanism of converting a state diagram into implementation code has some similarity with the State pattern [29], but State pattern neither addresses the issue of state hierarchy nor does it address concurrency within state diagrams. Joung et al. [30] show how icons can be added to the state diagrams and then code can be generated that animates the system.

Our earlier paper [15] demonstrates how dynamic model, represented as a single state diagram, can automatically be converted into Java code. The other paper [17] includes treatment of concurrent states within state diagrams. The present study focuses on generating code from the complete dynamic model that is represented by multiple state diagrams and activity diagrams. It also deals with active objects, which have their own thread of control.

11 Conclusions

A new method has been proposed to implement the dynamic behavior of an application, which is represented as a set of state transition diagrams and activity diagrams. To implement a state diagram, an object-oriented approach has been used where each state becomes a class and each transition becomes an operation. Inheritance mechanism is used to implement OR-type state hierarchy and the mechanism of object composition is used to represent AND-type state hierarchy. Activity diagrams, which represent the behavior of active objects, are implemented as Java threads. The method deals with intra-object concurrency (within a single object) and multiple thread concurrency (among several objects). The method has been implemented in our system, dCode, which automatically generates Java code from the object diagram, state diagrams and activity diagrams of a system. The code generated by dCode is approximately five times more compact and 38% more efficient than that of Rhapsody.

References

1. G. Booch, *"Object Oriented Design with Applications"*, Benjamin/Cummings, Redwood, California, 1991.
2. P. Coad and E. Yourdon, *"Object-Oriented Analysis"*, Prentice Hall, Eaglewood Cliffs, New Jersey, 1991.
3. I. Jacobson, *"Object-Oriented Software Engineering: A Use Case Driven Approach"*, Addison Wesley, Reading, Massachusetts, 1992.
4. P. Desfray, *"Object Engineering: The Fourth Dimension"*, Addison Wesley, Reading, Massachusetts, 1994.
5. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object-Oriented Modeling and Design", Prentice Hall, Eaglewood Cliffs, New Jersey, 1991.
6. Rational Software Corporation, Rational Rose, http://www.rational.com.
7. Object Domain Systems, Object Domain, http://www.object-domain.com/.

8. Excel Software, MacA&D, http://www.excelsoftware.com/index.html.
9. T. Kim, "Object Oriented Designer", http://www.qucis.queensu.ca/Software-Engineering/blurb/ OOD.html, ftp.x.org.
10. MicroGold Software, NJ, 08807, StateMaker, http://www.worldwidemart.com/mattw/software/Windows3.X/demo/
11. ObjectTime Limited, ROOM, http://www.objectime.on.ca/.
12. Advanced Software Technologies, Graphical Designer, http://www.advancedsw.com/.
13. Aonix, StP: Software Trough Pictures, http://www.ide.com/index.html.
14. i-Logix Inc., Rhapsody, http://www.ilogix.com.
15. J. Ali and J. Tanaka, "Automatic Code Generation from the OMT-based Dynamic Model", *In Proceedings of the Second World Conference on Integrated Design and Process Technology*, volume 1, pages 407-414, Austin, Texas, December 1996.
16. J. Gosling, B. Joy, and G. Steele, *"The Java Language Specification"*, Addison Wesley, Reading, Massachusetts, 1996.
17. J. Ali and J. Tanaka, "Generating Executable Code from the Dynamic Model of OMT with Concurrency", *In Proceedings of the IASTED International Conference on Software Engineering (SE'97)*, pages 291-297, San Francisco, California, USA, November 1997.
18. IEEE Computer Society, "Problem Set for the Fourth International Workshop on Software", *Specification and Design*, April 1987.
19. J. Rumbaugh, "Controlling Code", *Journal of Object-Oriented Programming*, 6(2): 25 -30, May 1993.
20. J. Rumbaugh, "Objects in the Twilight Zone", *Journal of Object-Oriented Programming*, 6(3): 18 -23, June 1993.
21. D. Harel, "Statecharts: A Visual Formalism for Complex systems", *Science of Computer Programming*, (8): 231 -274, August 1987.
22. D. Harel, "On Visual Formalisms", *Communications of the ACM*, 31(5): 514 -530, May 1988.
23. D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Transactions on Software Engineering and Methodology*, 5(4): 293 -333, October 1996.
24. Rational Software Corporation, "Unified Modeling Language (UML)", http://www.rational.com.
25. 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 Generation of Design Schema from C++ Program", *In Object-Oriented Symposium*, pages 111 -118, Tokyo, Japan, 1996. (In Japanese).
26. S. Nakashima, J. Ali, and J. Tanaka, "An Automatic Layout System for OMT-based Object Diagram", *In Proceedings of the Second World Conference on Integrated Design and Process Technology*, volume 2, pages 82 -89, Austin, Texas, December 1996.
27. D. Harel and E. Gery, "Executable Object Modeling with Statecharts", *Computer*, 30(7): 31-42, 1997.
28. D. Harel and E. Gery, "Executable Object Modeling with Statecharts", *In Proceedings of 18th International Conference on Software Engineering*, pages 246 –257, IEEE, March 1996.
29. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison Wesley, Reading, Massachusetts, 1995.
30. S. Joung, J. Ali, and J. Tanaka, "Automatic Animation from the Requirements Specifications based on Object Modeling Technique", *In Proceedings of International Symposium on Future Software Technology (ISFST-97)*, pages 133 -139, Xiamen, China, October 1997.

Jauhar Ali

Jauhar Ali is currently a chief engineer at the Software Engineering Laboratory, Software Research Associates (SRA) Inc., Japan. Before joining SRA, he worked as a foreign researcher at the Institute of Information Sciences and Electronics, University of Tsukuba. His research interests include object-oriented methodologies, code generation and reverse engineering. He is also interested in program visualization.

Ali received a BSc in 1986 and a MSc in 1990 from the University of Peshawar. He received a PhD in computer science in 1998 from the University of Tsukuba.

Jiro Tanaka

Jiro Tanaka is a professor in the Institute of Information Sciences and Electronics at the 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.

Tanaka 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.