Automatic Code Generation from Object Oriented Models

July 1998

Jauhar Ali

博士（工学）学位論文

Automatic Code Generation from Object Oriented Models

オブジェクト指向モデルからの自動コード生成

筑波大学大学院博士課程

工学研究科 電子・情報工学専攻

アリ ジョハル

寄

平成10年7月

99311258

Automatic Code Generation from Object Oriented Models

Jauhar Ali

July 1998

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering

Institute of Information Sciences and Electronics

Doctoral Program in Engineering

University of Tsukuba, Japan

Abstract

Object-oriented methodologies suggest to create different models of a system indicative of its different aspects. Object Modeling Technique (OMT), an object-oriented methodology, represents the static structure of classes in a system by an ER-style diagram called the object model, and the dynamic behavior of the classes by a set of state transition diagrams called the dynamic model. To speed up the software development process, it has been desired to have CASE tools that can automatically generate code from these models.

In the present work, an attempt has been made to find methods to automatically generate executable code from the object-oriented models in general and the dynamic model in particular. An object-oriented approach has been proposed to convert a state diagram representing the behavior of a multi-state class into code. States are treated as classes and transitions as their operations. *Inheritance* is used to implement state hierarchy and *object composition* is used to implement concurrent states. It is realized that active objects can well be represented by activity diagrams rather than state diagrams. In the proposed approach, active objects are implemented as Java threads.

A system, O-Code, has also been developed that implements the proposed method and automatically generates executable Java code from the specifications of the object and dynamic models. A comparison with Rhapsody shows that the code generated by O-Code is much more compact, efficient and understandable than that of Rhapsody.

Contents

List of Figures **6**

List of Tables **8**

1 Introduction **9**

1.1 Project Background 10

1.2 Automatic Code Generation 11

1.3 Goal and Objectives 12

1.4 Organization 12

2 Background **13**

2.1 Object Modeling Technique (OMT) 13

2.1.1 Three Models 14

2.2 Unified Modeling Language (UML) 16

2.3 Specification Languages (RSL and DSL) 17

3 Dynamic Model with a Single State Diagram **18**

3.1 Introduction 18

3.2 Converting a State Diagram into Code 18

3.2.1 Dealing with State Hierarchy 20

3.3 A Simple Calculator 21

3.3.1 Calculator: Object Model 22

3.3.2 Calculator: Dynamic Model 22

3.4 Automatic Code Generation 23

3.5 Executing the Generated Code 27

3.6 Some Other Cases 29

3.6.1 Transitions without Events 29

3.6.2 Transitions with Conditions 29

4 Dynamic Model with Concurrency 31

4.1 Introduction 31

4.2 Air Condition System 32

4.3 Implementing a State Diagram 34

4.3.1 Treatment of Concurrency 35

4.4 Automatic Code Generation 37

4.5 Executing the Generated Code 41

5 Dynamic Model with Multiple State Diagrams and Activity Charts 43

5.1 Introduction 43

5.2 The Elevator Example 44

5.3 Designing the Elevator System 44

5.4 The Controller Class 46

5.5 State Diagrams 47

5.5.1 Converting a State Diagram into Code 47

5.5.2 Optimizing the Code 49

5.6 Activity Diagrams 50

5.7 Classes having both State and Activity Diagrams 53

5.8 Automatic Code Generation 54

5.8.1 Generating Code for Domain Classes 55

5.8.2 Generating Code for State Classes 56

6 Automatic Code Generating System: O-Code 58

6.1 Introduction 58

6.2 Transformer 59

6.2.1 The readFile Method 60

6.2.2 The interpretFile Method 61

6.2.3 The interpretLine Method 61

6.3 Optimizer 62

6.3.1 The setDefaultStates Method 62

6.3.2 The findEventsActions Method 62

6.4 Code Generator 63

7 Discussion **64**

7.1 Discussion 64

7.2 Comparison with Rhapsody 66

7.2.1 Executing Image of the Code Generated by Rhapsody . . . 66

7.2.2 Executing Image of the Code Generated by O-Code 69

7.2.3 Comparing the Code Generated by Rhapsody and O-Code 69

8 Related Work **72**

8.1 Code Generating CASE Tools 72

8.2 Implementing State Diagrams 73

8.3 Other 74

9 Conclusions **75**

Acknowledgments **76**

Bibliography **78**

List of Figures

1.1 Overview of the HITO project 11

2.1 Overview of OMT object model notation 15

2.2 Overview of OMT dynamic model notation 16

3.1 Traditional approach to implement a state diagram 19

3.2 Proposed approach to implement a state diagram. 20

3.3 A simple calculator 21

3.4 Object model of the calculator 22

3.5 Dynamic model of the calculator 23

3.6 Dynamic model of the calculator in DSL format 24

3.7 Table representation of the state diagram for the calculator 25

3.8 Class structure of the generated code for the calculator 26

3.9 Transition without an event 29

3.10 Transition with guard condition 30

4.1 Remote control device for air conditioner 32

4.2 Object model of the remote control device 33

4.3 State diagram of the Controller 33

4.4 Implementing a state diagram having state hierarchy 35

4.5 State diagram with concurrent states and its implementation in Java 36

4.6 State diagram of the Controller in DSL format. 37

4.7 Table created by O-Code for the state diagram of the Controller . 38

4.8 Part of the generated code for the air conditioner 39

4.9 Class structure of the generated code for the air conditioner . . . 40

4.10 Sequence of operations when the Mode button is pressed 42

5.1 Object model for the elevator simulation system 45

5.2 State diagram for Floor class 47

5.3 Class structure of the Floor class and its companion classes 48

5.4 Activity diagram for Elevator class 52

6.1 Overall structure of O-Code system 59

6.2 Classes that represents the elements of a state diagram 60

7.1 Implementing a state diagram using data values to represent states 65

7.2 A conceptual view of the execution sequence in code generated by Rhapsody and O-Code 68

List of Tables

7.1 Comparing the mechanisms used by Rhapsody and O-Code to implement a state diagram 67

7.2 Comparing the compactness of the code generated by Rhapsody and O-Code 70

7.3 Comparing the efficiency of the code generated by Rhapsody and O-Code 71

Chapter 1

Introduction

Interest in object-oriented software development has grown rapidly over the last few years. Object-oriented modeling and design is a new way of thinking about problems using models organized around real-world concepts. Object-oriented models are useful for understanding problems, communicating with application experts, preparing documentation, and designing programs and databases. A number of object-oriented methodologies [1, 2, 3, 4, 5] have been proposed that cover the analysis, design and implementation phases of software development. These methodologies suggest to create different models which show different aspects of a system during the analysis and design phases. The models are converted into code during the implementation phase.

The Object Modeling Technique (OMT) uses three kinds of models to describe a system: the *object model*, describing the objects in the system and their relationships; the *dynamic model*, describing the interactions among objects in the system; and the *functional model*, describing the data transformations of the system. The model that becomes the most important at the design and implementation stages is the dynamic model. Without the dynamic model, one does not know the *behavior* of classes mentioned in the object model.

OMT and other object-oriented methodologies describe in 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. A big problem in the development of a system through object-oriented 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. It would be ideal to have CASE tools that automatically generate or help to generate executable code from the models. Most of the present CASE tools [6, 7, 8, 9] generate only header files from the object model, which is comparatively easy due to its static nature. To get an executable system, however, a user has to implement the dynamic and other models of the system and combine the implementation code with the header files.

1.1 Project Background

The present work is part of the HITO project which aims at automation of object-oriented software development. The OMT methodology [1] has been used in HITO project due to its support for rich graphical notation and consistency in models throughout the analysis, design and implementation phases of software development. Figure 1.1 shows the overview of the project. In HITO project, a series of tools have been developed which support and automate object-oriented analysis, design and implementation. Requirement Specifications List (RSL) language and Design Schema List (DSL) language have been designed and used for data exchanges among the tools [10]. The Automatic Modeling System [11], parses problem statement in a natural language and extracts model elements, e.g., classes, associations, events etc.

The sub-group of HITO project at IP-Lab, University of Tsukuba, has concentrated research on automatic layout of models and automatic code generations from object-oriented models. The dashed-line box in Figure 1.1 represents the work carried out at the University of Tsukuba. The Automatic Layout System [12, 13] uses graph layout algorithms and automatically calculates a readable layout from the model elements in RSL format. The output of the automatic layout system is in DSL format used by other tools as input. The Code Generating

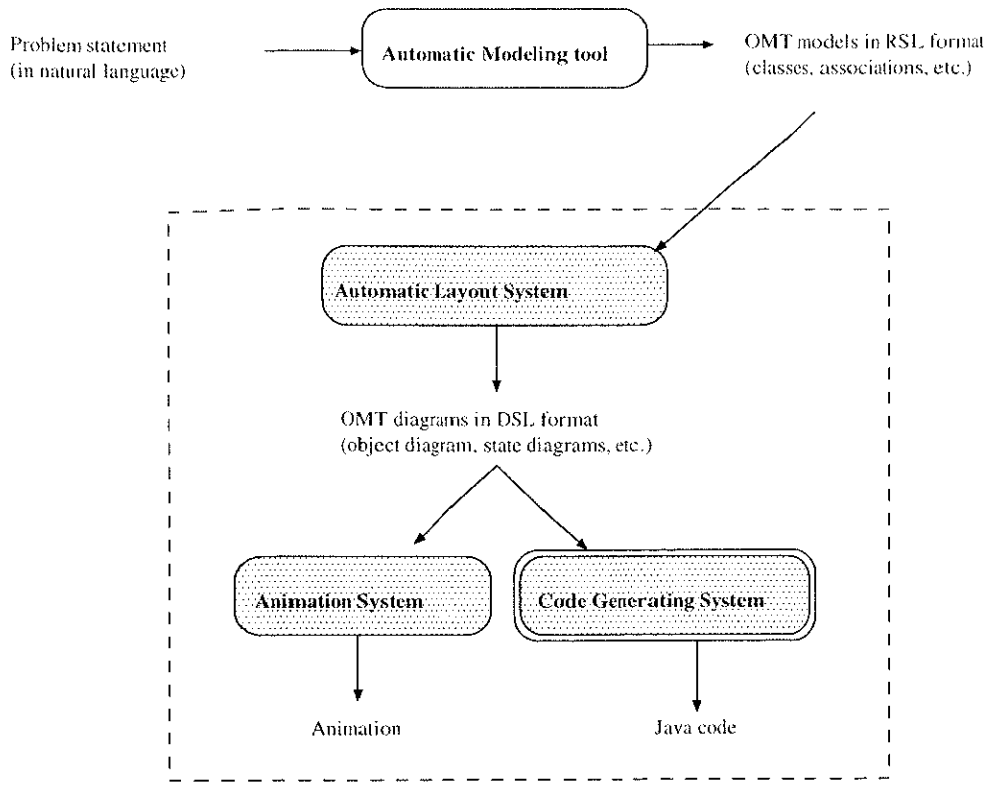


Figure 1.1: Overview of the HITO project

System [14, 15, 16, 17, 18] generates executable Java code from OMT models in DSL format. The Animation System [19] produces animation from OMT models.

1.2 Automatic Code Generation

The present study reports that part of the project which covers automatic code generation from object-oriented models. An attempt has been made to find methods to automatically generate implementation code from object-oriented models in general and the dynamic model in particular.

1.3 Goal and Objectives

The final goal of this research is to automatically generate efficient implementation code from the object-oriented models. The general objectives are:

1. To find methods to generate efficient and easier to understand implementation code from the dynamic model in an object-oriented language like Java.

2. To implement the proposed methods and develop a system to automatically generate executable Java code from the object and dynamic models.

1.4 Organization

This thesis is organized as follows. Chapter 2 gives the fundamental knowledge and definitions of the terms used in this work. Chapter 3 shows how code can be generated from the dynamic model that is represented with a single state diagram. Chapter 4 describes the treatment of concurrent states in state diagrams and intra-object concurrency. Chapter 5 discusses implementation of the dynamic model with many state diagrams. Active objects and multiple-thread concurrency are also discussed. Chapter 6 explains the automatic code generating system, O-Code. In Chapter 7, a discussion about the proposed approach is given. This Chapter also gives a comparison of the developed system to Rhapsody. Chapter 8 presents the related work. Finally, in Chapter 9, the main results of our research are summarized.

Chapter 2

Background

We have used the graphical notation of Object Modeling Technique (OMT), and upto some extent Unified Modeling Language (UML), in various diagrams. Also the input to our code generating system is in Design Schema List (DSL) language format. This chapter gives a short description of OMT, UML and DSL.

2.1 Object Modeling Technique (OMT)

Object Modeling Technique (OMT) [1, 20, 21, 22, 23, 24, 25, 26, 27] is an object-oriented software development methodology which extends from analysis through design to implementation. First an analysis model is built to abstract essential aspects of the application domain without regard to eventual implementation. This model contains objects found in the application domain, including a description of the properties of the objects and their behavior. The design decisions are made and details are added to the model to describe the implementation. Finally the design model is implemented in a programming language.

OMT describes a graphical notation for expressing object-oriented models. Application-domain and computer-domain objects can be modeled using the same notation. A great advantage of OMT is that the same seamless notation is used

from analysis to design to implementation so that information added in one stage of development need not be lost or translated for the next stage.

2.1.1 Three Models

The OMT methodology uses three kinds of models to describe a system: the *object model*, describing the objects in the system and their relationships; the *dynamic model*, describing the interactions among objects in the system; and the *functional model*, describing the data transformations of the system.

The Object Model

The *object model* describes the static structure of the objects in a system and their relationships. The object model contains object diagrams. An *object diagram* is a graph whose nodes are *classes* and whose arcs are *relationships* among classes. Figure 2.1 summarizes the notation used in the object model.

The Dynamic Model

The *dynamic model* describes the aspects of a system that change over time. The dynamic model is used to specify and implement the *control* aspects of a system. The dynamic model contains state diagrams. A *state diagram* is a graph whose nodes are *states* and whose arcs are *transitions* between states caused by *events*. A state diagram may contain nested states or substates. In the case of OR-type substates, only one of the substate can be active at a given time when their superstate is active. In the case of AND-type substates, all the substates become active simultaneously whenever their superstate gets activated. An *entry* action is executed whenever the corresponding state is entered. It is indicated by writing the action-name after the string "entry/" inside the state node. Similarly, an *exit* action is executed whenever the corresponding state is exited. It is indicated by writing the action-name after the string "exit/" inside the state node. *Internal*

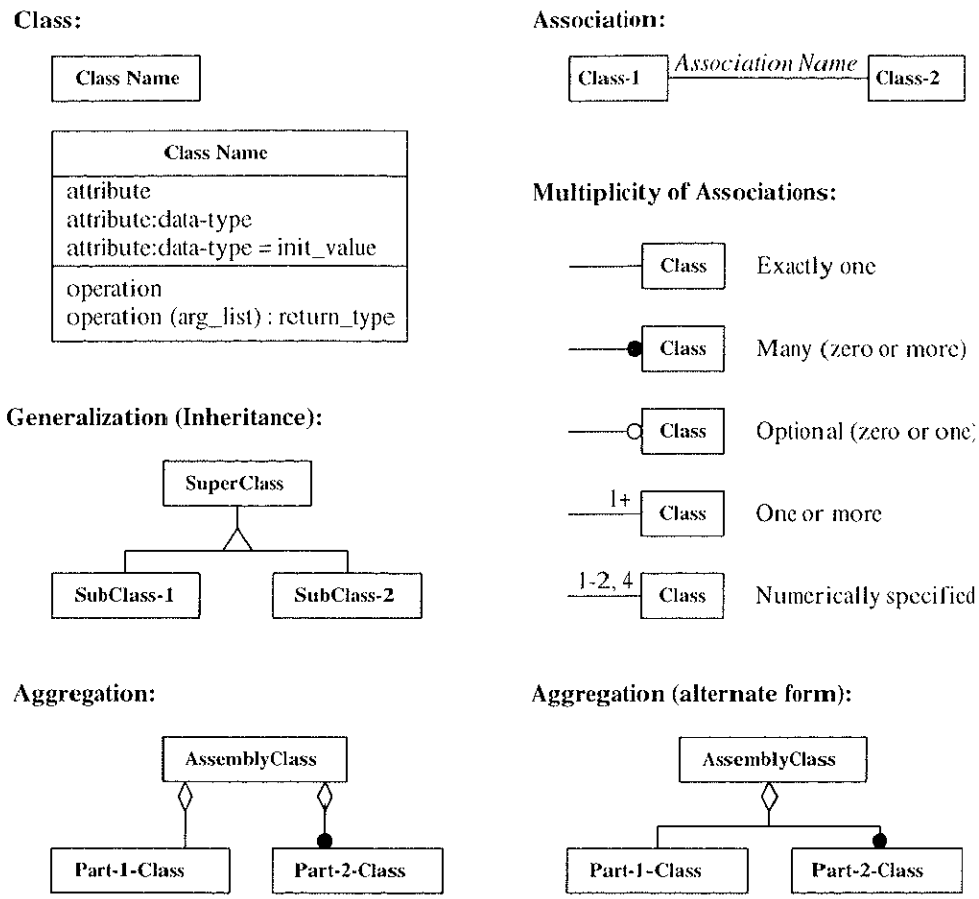


Figure 2.1: Overview of OMT object model notation

events cause some actions to be executed without changing the state. Such events are written inside the state along with the corresponding action separated by a slash. Figure 2.2 summarizes the notation used in the dynamic model.

The Functional Model

The *functional model* describes the data value transformations within a system. The functional model contains data flow diagrams. A data flow diagram represents a computation. A *data flow diagram* is a graph whose nodes are *processes* and whose arcs are *data flows*.

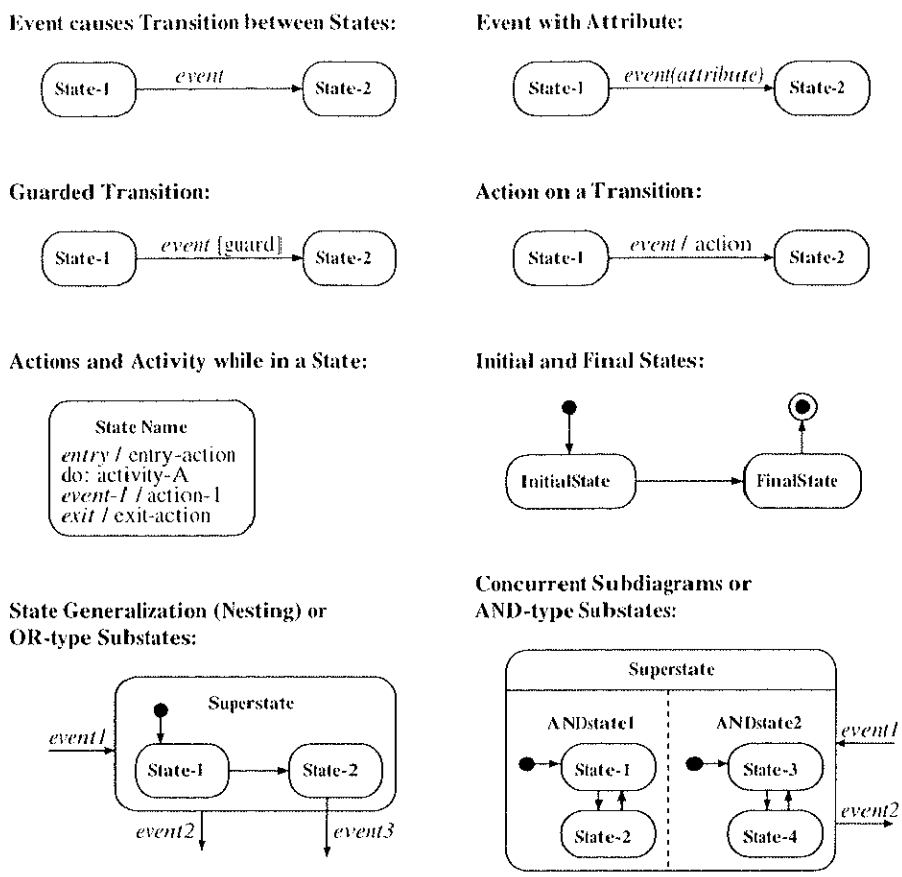


Figure 2.2: Overview of OMT dynamic model notation

2.2 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the successor to the wave of object-oriented analysis and design (OOA&D) methods that appeared in the late '80s and early '90s. It unifies the methods of Booch [2, 28], Rumbaugh (OMT), and Jacobson [4]. The UML is called a modeling language, not a method. Most methods consist, at least in principle, of both a modeling language and a process. The *modeling language* is the (mainly graphical) notation that methods use to express designs. The *process* is their advice on what steps to take in doing a design. UML uses many extra diagrams, in addition to the diagrams of OMT. One of the diagrams which interests us is the activity diagram. An *activity diagram* is a special case of a state diagram in which states are action states (activities) and in which transitions are triggered by completion of the actions in

the source states.

2.3 Specification Languages (RSL and DSL)

Requirements Specification List (RSL) language and Design Schema List (DSL) language [10] are specification languages designed to facilitate data exchanges among tools. RSL describes OMT models in an abstract way without any graphical information. It only gives the model elements that form a model. For example, the names of the classes, the constituent classes of an association, etc. DSL, on the other hand, describes the diagrams representing the OMT models in full detail. It includes the graphical information of the model elements. For example, the location and size of the node that represents a class, the end points of the line that represents an aggregation, etc.

Chapter 3

Dynamic Model with a Single State Diagram

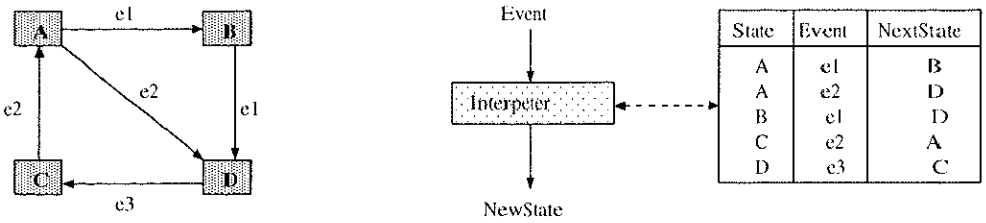
3.1 Introduction

Though a real system has a dynamic model with many state diagrams, the behavior of a small system can be represented by a single state diagram [14]. As described by Rumbaugh [20, 21, 26], there is normally a *controller* class that represents the entire system and keeps the main flow of control. Our investigation has made us to believe that when the dynamic model is represented by a single state diagram, the state diagram actually represents the behavior of the controller class. Users interact with the system through its user interface component, which sends messages to the controller class. The messages become events for the controller and it responds accordingly.

3.2 Converting a State Diagram into Code

To automatically generate implementation code from the dynamic model, first we looked for methods that are used to convert a state diagram into software

code. The traditional approach to implement a state diagram is to represent it as a table and write an interpreter to execute it [29, 20] (Figure 3.1). The table consists of a list of states, each of which has a table of transitions. Each time an event occurs, the interpreter searches the current state for a transition on that event. If a transition is found, it is executed and the current state is changed, otherwise the event is ignored. The traditional approach is suitable for simple state diagrams which do not contain state hierarchy.



(a) State Diagram.

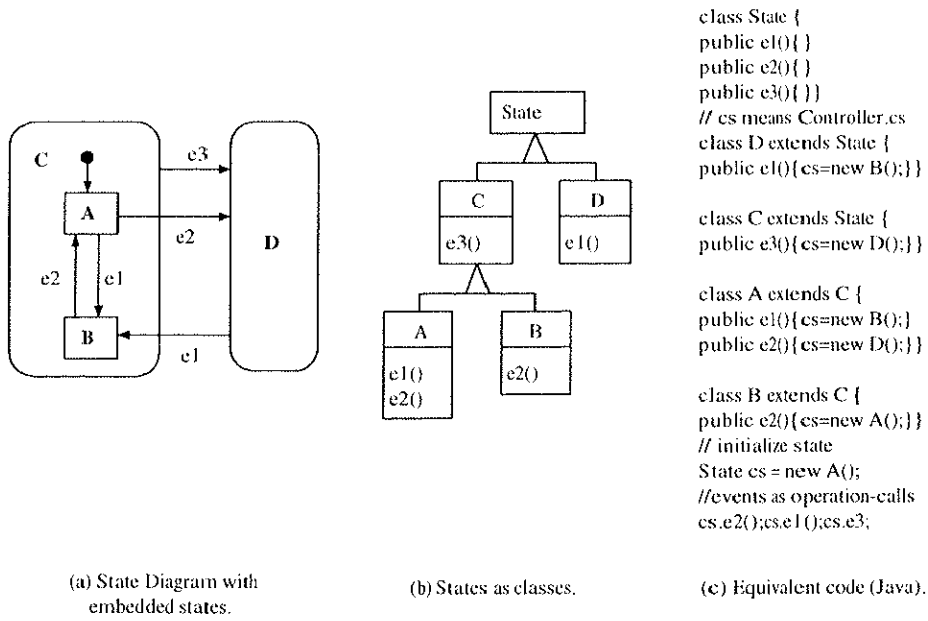
(b) Table-interpreter implementation.

Figure 3.1: Traditional approach to implement a state diagram

Our objective is to find a method to convert the state diagram into actual code that can directly be executed without the need of an interpreter. We propose an object oriented approach to implement a state diagram. For each state, a class is created that implements the state-specific behavior. We call such classes as state classes. The key idea in our approach is to introduce an abstract class (say State) to represent the states of the domain class (say Controller) which the state diagram belongs to. The 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 state diagram. Normally, there is only one object of one of the state classes that is active at a given time and represents the current state when the application is running. This object is pointed to by an attribute (say cs) in the Controller class, which represents the entire application. Each time an event occurs, it is translated into an operation call on the current state object (cs).

3.2.1 Dealing with State Hierarchy

In state diagrams, there are often substates within superstates, where the superstates have transitions that are common to their substates. We observed that the state hierarchy in a state transition diagram resembles to the class hierarchy (inheritance tree) in an object-oriented program. In the former, substates inherit the behavior of their super-state, while in the latter, subclasses inherit the behavior of their parent class. In our approach, where each state of a multi-state class is implemented as a class, substates become subclasses of the class for the superstate (Figure 3.2). The superclass implements the behavior specific to the superstate and the subclasses implement the behavior specific to the substates. Thus by using the inheritance mechanism, we became able to extend our model for implementing state diagrams to also accommodate state hierarchy. Here we restrict our discussion to only OR-type substates.



(a) State Diagram with embedded states.

(b) States as classes.

(c) Equivalent code (Java).

Figure 3.2: Proposed approach to implement a state diagram.

3.3 A Simple Calculator

We use an example to demonstrate our approach. Suppose we want to develop a program for a simple interactive calculator. Figure 3.3 shows the appearance of the calculator. We suppose that the calculator has the following functions:

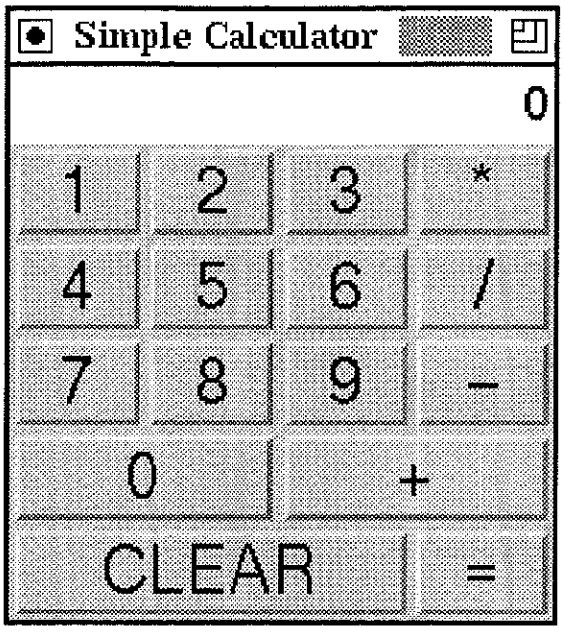


Figure 3.3: A simple calculator

1. Simple calculations; $10 + 25 = 35$.

2. Continuous calculations involving many operators; $10 * 3 - 5 + 10 = 35$.

3. After getting an answer by clicking on =, any subsequent click on = repeats the previous operation, using the *result* as *operand1*; $10 + 4$ ===== generates the sequence 14 18 22 26 30.

4. A calculation can be aborted by clicking on CLEAR, a fresh calculation can then be started by entering *operand1*.

3.3.1 Calculator: Object Model

Figure 3.4 shows the object model for this application. Display and ButtonPanel classes are directly related to the Controller class. Display represents the small display area of the calculator where the values are being entered and the computed result is shown. ButtonPanel represents the remaining part of the calculator containing various buttons, which the user clicks while using the calculator. After receiving the *setValue* message from the Controller, the Display object changes its value. The ButtonPanel object is an aggregation of various Button objects, each of which has a name. When a button is clicked by the user, it becomes an event for the system. The ButtonPanel object then sends an appropriate message to the Controller.

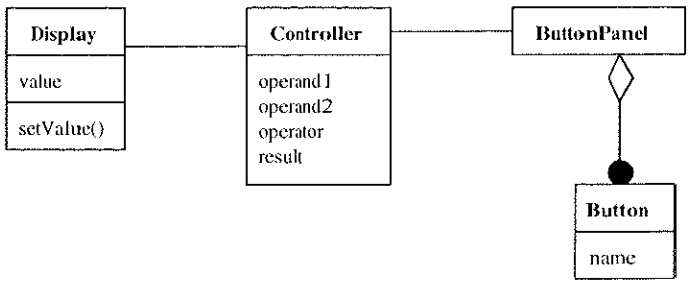


Figure 3.4: Object model of the calculator

3.3.2 Calculator: Dynamic Model

The class having the most important dynamic behavior is the Controller. It has four attributes, namely operand1, operand2, operator and result. These attributes define the state of the Controller. The Controller behaves differently in different states. Figure 3.5 shows the complete behavior of the Controller in the form of a state diagram. In this diagram, possible events for the system are *digit(n)*, *operator(c)*, *equal* and *clear*. Parameter n can take a digit character (0 to 9) and parameter c can take one of the four operator symbols (+, -, *, /). These events come as messages from the ButtonPanel object whenever some button is clicked.

To keep the names of actions (and corresponding methods) simplified, we have used O1 for operand1, O2 for operand2, Op for operator and R for result.

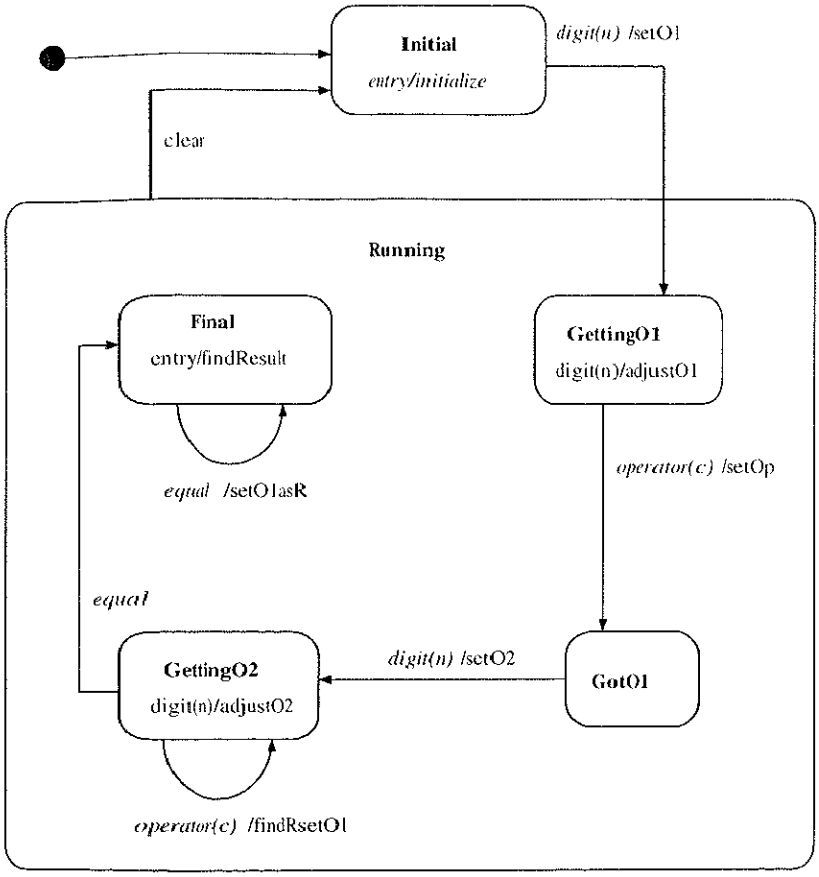


Figure 3.5: Dynamic model of the calculator

3.4 Automatic Code Generation

We have developed a system that implements the proposed method and automatically generates executable Java code [30] from the dynamic model that consists of a simple state diagram. The input to the system is a specification of the state diagram in Design Schema List (DSL) language. Figure 3.6 shows the dynamic model of the calculator in DSL format. The system works in two steps. First, it reads the DSL file and identifies various components of the state diagram. It makes a table and records all the information properly, such as various states,

their substates, events and transitions. Figure 3.7 shows the table created by the system from the DSL file of the calculator (Figure 3.6). In the second step, the system takes information from the table and generates Java code following our proposed method of converting a state diagram into code. Figure 3.8 shows the class structure of the generated code. The system follows the following rules to convert a state diagram into code:

```
OSTD(g1)[nodes{n1,n2,n3,n4,n5,n6},arcs{a1,a2,a3,a4,a5,a6,a7}];

OSTDN(n1)[loc(140:50),size(60:30),
ostdnAttr(name:Initial,entry/initialize)];
OSTDN(n2)[loc(50:100),size(220:180),
ostdnAttr(name:Running,substates{n3,n4,n5,n6})];
OSTDN(n3)[loc(190:130),size(60:30),ostdnAttr(name:GettingO1,
event(name:digit,arg(char:n))/adjustO1)];
OSTDN(n4)[loc(200:220),size(40:30),ostdnAttr(name:GotO1)];
OSTDN(n5)[loc(80:220),size(60:30),ostdnAttr(name:GettingO2,
event(name:digit,arg(char:n))/adjustO2)];
OSTDN(n6)[loc(80:130),size(60:30),
ostdnAttr(name:Final,entry/findResult)];

OSTDA(a1)[from(n1,side:RIGHT,off:15),to(n3,side:TOP,off:40),
ostdaAttr(name:digit,arg(char:n)/setO1)];
OSTDA(a2)[from(n3,side:BOTTOM,off:20),to(n4,side:TOP,off:15),
ostdaAttr(name:operator,arg(char:c)/setOp)];
OSTDA(a3)[from(n4,side:LEFT,off:15),to(n5,side:RIGHT,off:15),
ostdaAttr(name:digit,arg(char:n)/setO2)];
OSTDA(a4)[from(n5,side:LEFT,off:15),to(n6,side:LEFT,off:15),
ostdaAttr(name:equal)];
OSTDA(a5)[from(n6,side:BOTTOM,off:15),to(n6,side:BOTTOM,off:45),
ostdaAttr(name:equal/setO1asR)];
OSTDA(a6)[from(n2,side:TOP,off:30),to(n1,side:LEFT,off:22),
ostdaAttr(name:clear)];
OSTDA(a7)[from(n5,side:BOTTOM,off:15),to(n5,side:BOTTOM,off:45),
ostdaAttr(name:operator,arg(char:c)/findRsetO1)];
```

Figure 3.6: Dynamic model of the calculator in DSL format

1. A class is defined for each state. The name of the state becomes the name of the class. If this state is a substate of another state then the class becomes a subclass of the class for that superstate. Otherwise, it is subclassed from the ControllerState class, which is an abstract class for all state classes.

2. Each event on a state becomes a method in the corresponding class. The method has the same name as the event. If the event has parameters, the corresponding method is also defined with parameters. Body code for the method is also generated. If the event is an *internal* one, the body code

State ID	State Name	Entry	Super-state	Sub-states	Transitions						Internal Events			
					ID	Event			Action	Next State	Event			Action
						Name	Argument				Name	Argument		
							Name	Type				Name	Type	
n1	Initial	initialize			a1	digit	n	char	setO1	n3				
n2	Running			n3 n4 n5 n6	a6	clear				n1				
n3	GettingO1		n2		a2	operator	c	char	setOp	n4	digit	n	char	adjustO1
n4	GotO1		n2		a3	digit	n	char	setO2	n5				
n5	GettingO2		n2		a4	equal				n6	digit	n	char	adjustO2
					a7	operator	c	char	findRsetO1	n5				
n6	Final	findResult	n2		a5	equal			setO1asR	n6				

Figure 3.7: Table representation of the state diagram for the calculator

just contains a method call which executes the action associated with that internal event. If the event has a transition, the body code contains the following:

(a) A method call to execute the *exit* action of the current state.

(b) A method call to execute the action related to the transition.

(c) Code for deleting the current state object and creating an object of the new state.

(d) A method call to execute the *entry* action of the new state.

If the event contains parameters, these are passed to the methods of the corresponding actions when they are called. The above code of (a) to (d) comprises the *transition code*. From now on, we shall refer to it by transition code. In Figure 3.8, to save space the methods that implement transitions are indicated by a right-headed arrow mark. Code for one such method, operator(char c) in class Getting01, is shown in a separate box.

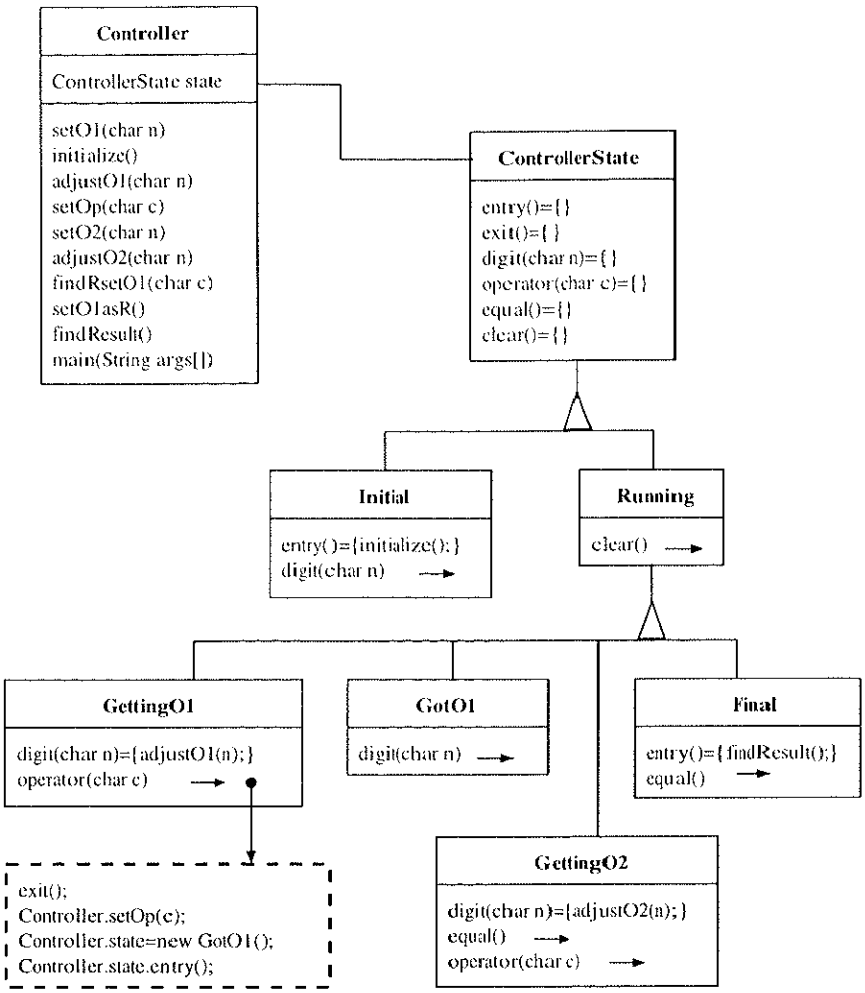


Figure 3.8: Class structure of the generated code for the calculator

3. If a state has *entry/exit* actions, then methods having the names entry and exit, respectively, are defined in the corresponding class. Bodies of these methods just contain a method call to the corresponding entry/exit actions.

4. All actions in the state diagram become methods in the Controller class. If their corresponding events have parameters, then these methods are defined with those same parameters. No code is generated for the bodies of these methods. A user has to insert implementation code to run the application. The reason for inserting all action operations in the Controller class is that these operations can contain messages to other objects of the application that are described in the object model. As the Controller class has references

to almost all important classes, it is the most suitable place.

5. ControllerState is an abstract class. This class is created only to provide a common interface to all state classes. It contains declarations of operations for all events in the state diagram. Entry and exit operations are also declared. No implementation code for operations is present here. Each subclass has its own implementation code for its own events. Due to the presence of the ControllerState class, the generated code is able to use *polymorphism* and select a method at runtime for execution when an event has occurred.

3.5 Executing the Generated Code

The code, generated by our system, does not represent the whole application. It represents that part of the system which is described by the dynamic model, i.e., control and sequences of operations. To execute this, we have to provide code for other parts of the system, particularly, code for the classes that are described by the object model. Internal code for the actions should also be provided since a dynamic model just contains names of the actions. In the remaining part of this section, we show how the code generated for the calculator example can be executed.

Figure 3.8 is the class structure of the code generated from the dynamic model of the system. The user does not have to care about this diagram. It is shown here to help understanding the generated code. This code represents the main flow of control in the application and can be executed by just providing the body codes for the methods of the Controller class and definitions of the Display and ButtonPanel classes. The body codes for other methods are already generated automatically.

The Controller class has an attribute state of the type ControllerState. The value of this attribute always shows the current state of the Controller. Whenever an event occurs, a message is sent to the object pointed to by the state variable.

If there is an implementation for the operation in the class (or superclass) of the object, it will be executed, otherwise the abstract method for that operation in the ControllerState class will get executed, which does nothing. To execute the generated code of this particular example, we proceeded as;

1. We defined Display and ButtonPanel classes, which came from the object model of the calculator. Using the Abstract Window Toolkit (AWT) classes of the Java language, these definitions were quite simple. In response to a button click, the ButtonPanel object sends a certain message to the Controller class. For example, if button "5" is clicked, it will send the message digit('5') to the object pointed to by Controller.state. The message becomes an event for the Controller.

2. We added four attributes in the Controller class, namely, operand1, operand2, operator and result, because they are present in the object model. In addition, we inserted an attribute display, which is a reference to the Display object, since there is an association between the Controller and the Display classes. Whenever the value of operand1, operand2 or result is changed, the Controller sends a message to the Display object to change its value.

3. We provided body code for all the methods of the Controller class. Except the main() method, all are actions in the dynamic model of Figure 3.5.

4. We wrote code for the main() method of the Controller. It just initializes the display and state attributes and activates a window to show the appearance of the calculator.

Note that to execute the application, almost all the changes are needed in the Controller class. The ControllerState class and its subclasses remain unchanged. Now as the Controller class contains the *main()* method, it can directly be executed by the Java interpreter. We run the program and it really worked in all the cases explained in Section 3.3.

3.6 Some Other Cases

There are some cases that are not covered in the calculator example, but our system can generate code for them. We want to mention these cases without giving complete examples, and show how our system will generate code for them.

3.6.1 Transitions without Events

Sometimes there are transitions that have no events. Such transitions are executed as soon as the current state finishes its own activity. If a state has such transition, the *transition code* is placed in the body of entry operation, after the method call to the entry action, as shown in Figure 3.9. In this way, the transition code is executed just after the state's entry action is completed.

Generated code for method
entry() in class *State1*

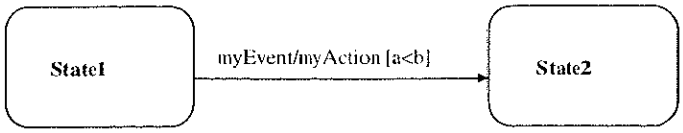
```
public void entry(){
  action1();
  state.exit();
  state = new State2();
  state.entry();
}
```

Figure 3.9: Transition without an event

3.6.2 Transitions with Conditions

There are some transitions that have a guard condition on them. Such transitions can only be executed if the associated event occurs and at the same time the associated condition is true. As shown in Figure 3.10, the entire *transition code*

for such transition is placed inside an *if statement*. So when the event occurs, the corresponding operation for that event will get executed but the transition will not be executed unless the condition is true.



Generated code for method
myEvent() in class *State1*

```
public void myEvent(){
  if (a<b){
    myAction();
    state.exit();
    state = new State2();
    state.entry();
  }
}
```

Figure 3.10: Transition with guard condition

Chapter 4

Dynamic Model with Concurrency

4.1 Introduction

As already described, the dynamic model is represented by a set of state transition diagrams each of which show the behavior of a particular class of objects. In the previous chapter, code generation from a simple state diagram without concurrency is discussed. A state diagram can contain concurrent states (also called AND-states) that become active simultaneously whenever their superstate gets activated. Concurrent states not only represent the inherent parallelism in some of the objects but also enable compact descriptions of complex state diagrams [31, 32]. This chapter addresses the issue of concurrency in state diagrams and how the proposed approach deals with concurrency while generating code from a state diagram [15, 16].

4.2 Air Condition System

We use an example throughout this chapter to simplify the explanation of our approach. Consider an Air Conditioner that is operated with a remote control device (Fig. 4.1). The remote device contains several buttons namely, Mode, Speed, 1, 2, 3, 4 and On/Off; and three rectangles which show the current state of the air conditioner. When the air conditioner is off, the display area (three rectangles) shows nothing. Suppose we develop software for this system.

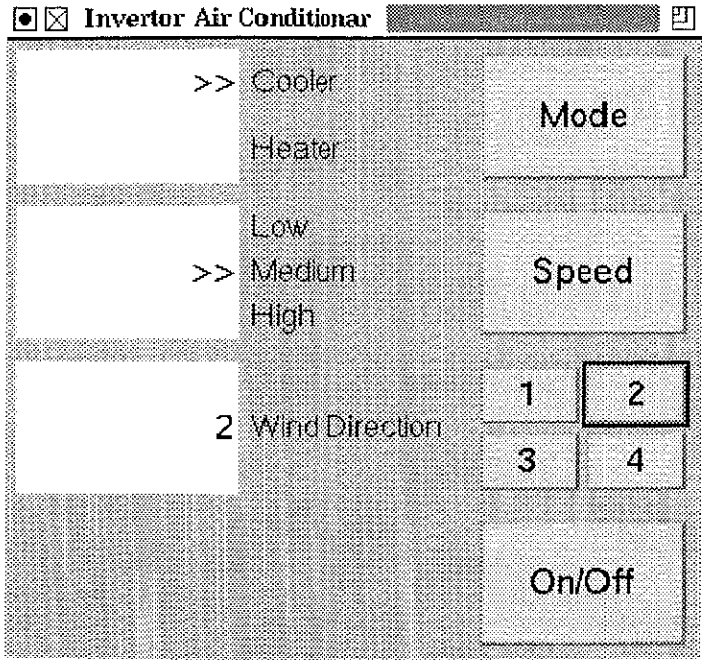


Figure 4.1: Remote control device for air conditioner

Figure 4.2 shows the object model for this application. The RemoteInterface class, which maintains one object instance of the DisplayArea class and seven object instances of the Button class, is directly related to the Controller class, which keeps control of the entire system [20, 21, 26]. Whenever some button is pressed, the Controller is informed by the RemoteInterface by sending a particular message. The response from the Controller depends on its current state. Figure 4.3 shows the behavior of the Controller in the form of a state diagram. Possible events (in-coming messages) for the Controller are: *onOffBut*, *modeBut*, *speedBut* and

dirBut(n), where parameter n can take an integer value from 1 to 4. The text on the right side of the slash (/) on a transition represents action that will get executed when the transition is fired.

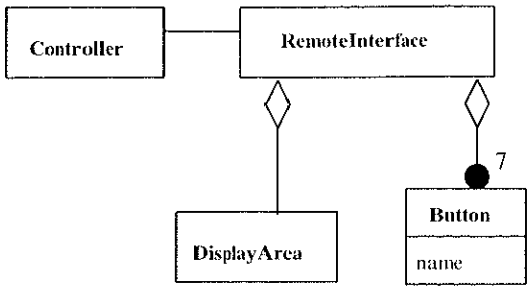


Figure 4.2: Object model of the remote control device

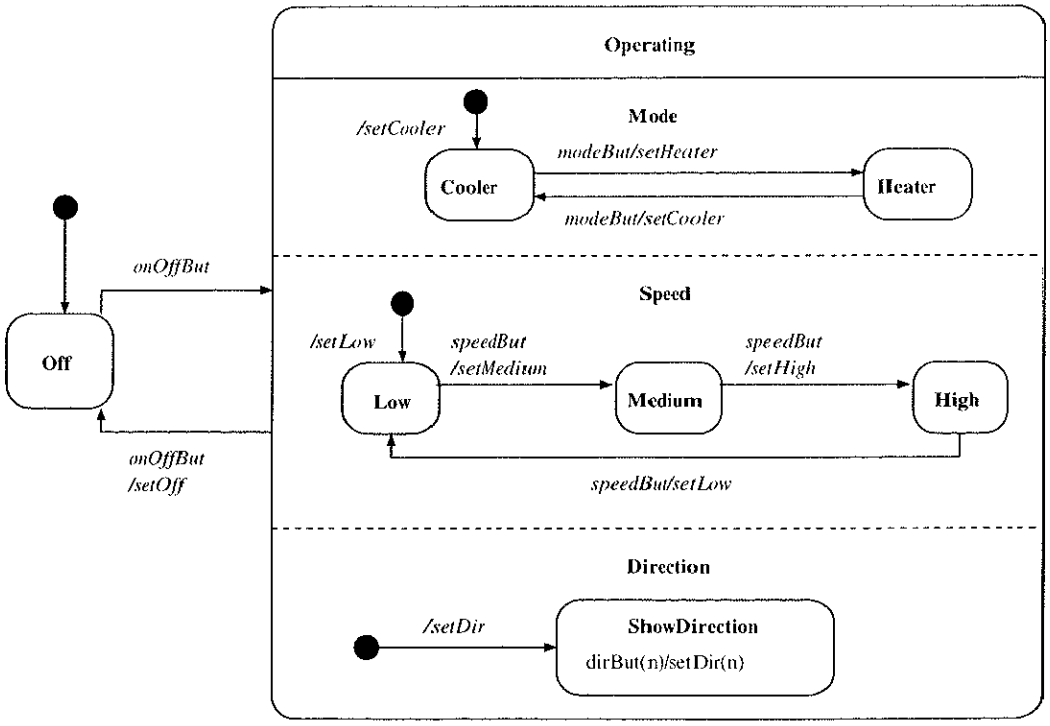


Figure 4.3: State diagram of the Controller

There are two possible states: Off and Operating. These states are activated alternatively whenever an *onOffBut* event occurs. The Operating state is a combination of three concurrent states or *AND-substates* [33, 34, 35]), namely Mode, Speed and Direction. So they all become active at the same time whenever the

Operating state gets activated. Each of the concurrent states has a number of *OR-substates* [33, 34, 35], e.g., Mode has substates Cooler and Heater. Only one of the OR-substates can be activated at a given time. A transition from a solid circle to a state shows that the state is the default one.

4.3 Implementing a State Diagram

As shown in the state diagram (Fig. 4.3), Controller responds to requests from other objects differently depending on its current state. For example, the effect of a modeBut request depends on whether the Controller is in Off state, Cooler state or Heater state.

Using our approach discussed in the previous chapter, we can only convert a state diagram that does not have concurrent states into implementation code. For example, Figure 4.4 shows a simplified version of Figure 4.3 (without concurrent states) along with the resulting class structure and Java code [30]. We use the prefix "C_" in class names representing the word "Controller". The class Controller maintains an attribute state of C_State type which points to an instance of a subclass of C_State and represents the current state of the Controller. The Controller delegates all state-specific requests to the state object. The state object performs operations particular to the current state of the Controller.

As can be seen in the implementation code (Fig. 4.4), each time the Controller changes state, the value of state gets changed. For example, when the Controller goes from Off to Cooler, an instance of the C_Cooler class will replace the instance of the C_Off class that was the current state object. The state object thus looks like changing its class. This is valid because of the substitution rule that allows a class instance to replace its superclass instance [36]. In other words, an instance of a subtype can always be used in any context in which an instance of a super type was expected.

Cooler and Heater are substates of Mode. When Mode is active, either Cooler

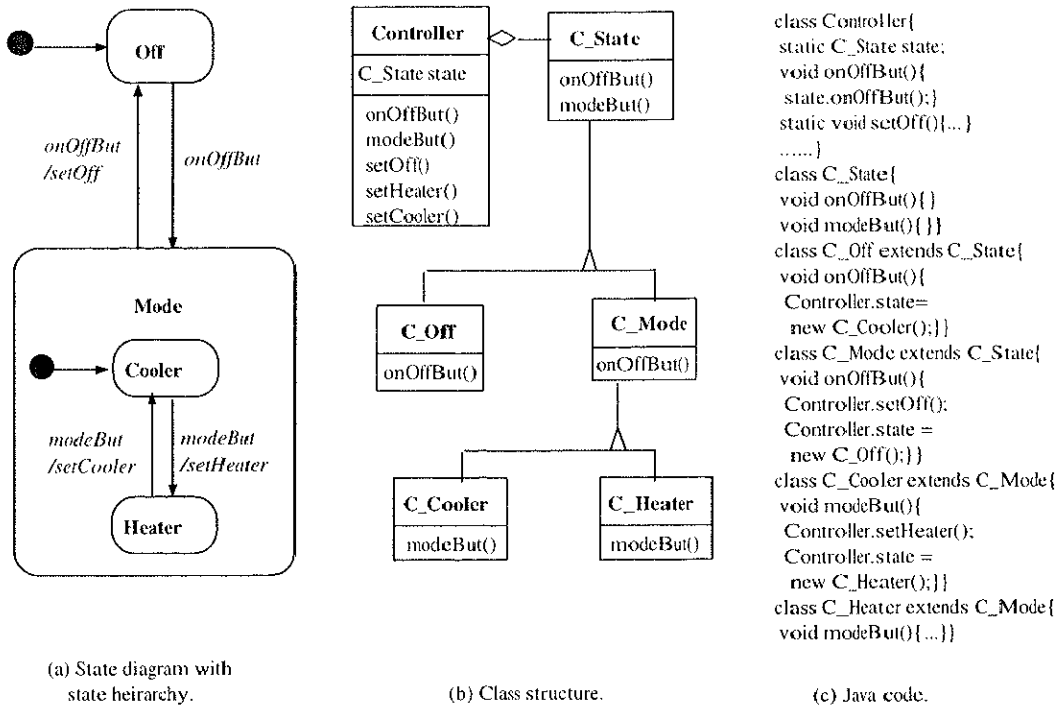


Figure 4.4: Implementing a state diagram having state hierarchy

or Heater will be active too. Both Cooler and Heater, when active, respond to the *onOffBut* event because there is a transition on this event from the superstate Mode. C_Cooler and C_Heater become subclasses of the C_Mode class. C_Mode class contains implementation for the onOffBut() operation which is inherited by both of its subclasses. The subclasses contain implementation for the operation modeBut().

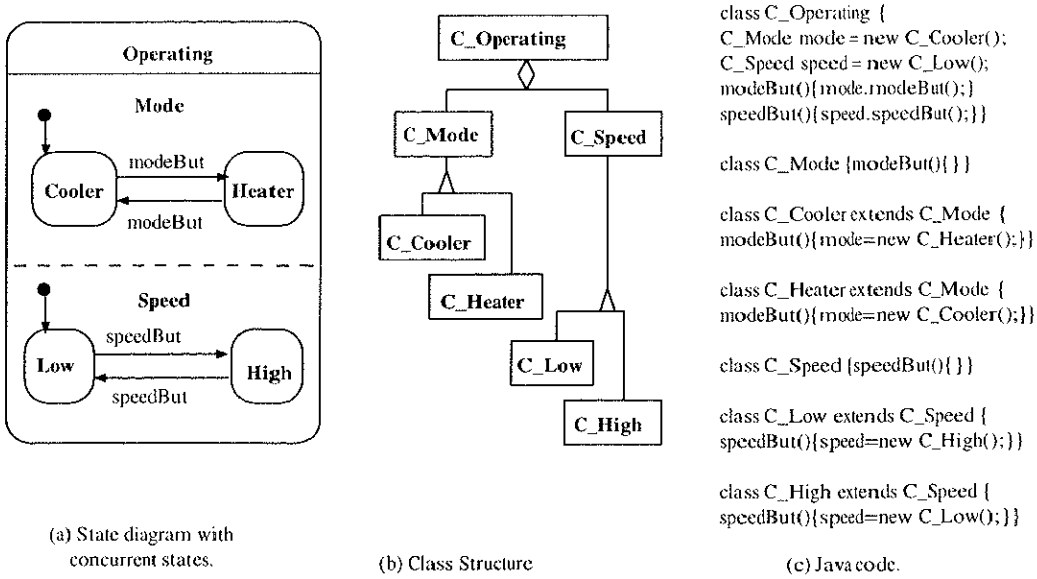
4.3.1 Treatment of Concurrency

As discussed earlier, we represent an active state 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 states whenever their superstate becomes active. That is why, we represent the superstate of AND-states as a *composite* class that owns objects of other classes.

The composite class maintains objects corresponding to each of the AND-

states. When the superstate of AND-states becomes active, the corresponding composite class gets instantiated. The composite object then instantiates its own state objects. The instantiation of the composite class thus guarantees the instantiation of the classes that correspond to the AND-states. 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-states at once when their superstate becomes inactive.

As an example, Figure 4.5 shows part of the state diagram of the air conditioner's Controller and the equivalent implementation code. C_Operating is a composite class and holds objects of type C_Mode and C_Speed classes. C_Mode and C_Speed classes, representing the two concurrent states, are abstract classes and serve as interface classes for their own subclasses. C_Cooler, C_Heater and C_Low, C_High become concrete subclasses of C_Mode and C_Speed respectively.



(a) State diagram with concurrent states.

(b) Class Structure

(c) Java code.

Figure 4.5: State diagram with concurrent states and its implementation in Java

As can be seen in the implementation code (Fig. 4.5), the composite class delegates the requests (events) on which there are transitions within the AND-states to the corresponding component state objects (e.g., modeBut is delegated to mode state object and speedBut is delegated to speed state object). For transitions

that are going out of the superstate of the AND-states (e.g., Operating), the composite class (C_Operating) provides the implementation code and does not forward them to the substate objects.

4.4 Automatic Code Generation

Our code generation system, O-Code, follows the above approach and automatically generates implementation code from the dynamic model. The input to the system is a specification of the dynamic model in Design Schema List (DSL) language [10] (Figure 4.6). After reading the DSL file, the system first identifies various states, their substates, transitions and internal events. It makes a table (Figure 4.7) to properly record all the information. The system then generates executable Java [30] language code (Figure 4.8) from the table. Figure 4.9 (solid lines portion) shows the class structure of the generated code for the dynamic model (Figure 4.3) of the air conditioner. The detailed rules for code generation are as follows:

```
OSTD(g1)[nodes{n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,n14,n15},
  arcs{a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11}];

OSTDN(n1){loc(57:150),size(10:10),ostdnAttr(name:START)};
OSTDN(n2)[loc(90:207),size(57:43),ostdnAttr(name:Off)];
OSTDN(n3)[loc(187:37),size(393:366),ostdnAttr(name:Operating,concurrent{n4,n8,n13})];
OSTDN(n4)[loc(187:67),size(393:96),ostdnAttr(name:Mode,substates{n5,n6,n7})];
OSTDN(n5)[loc(207:120),size(10:10),ostdnAttr(name:START)];
  ..........
OSTDA(a1){from(n1,side:BOTTOM,off:5),to(n2,side:TOP,off:28)};
OSTDA(a2)[from(n2,side:TOP,off:45),to(n3,side:LEFT,off:160),ostdaAttr(name:onOffBut/setOn)];
  ..........
```

Figure 4.6: State diagram of the Controller in DSL format.

1. The Controller class, with an attribute state of type C_State, is defined. All state-specific requests from other objects, i.e., the events in the state diagram, become methods in the Controller class. The body code for these methods just contain a method call to the same method in the state object, which means that the requests are delegated to the state object. All actions

State ID	State Name *=default	Substates *=concurrent	Transitions			Internal Event
			Event	Action	Next State	
n2	Off (*)		onOffButton		n3	
n3	Operating	n4,n8,n13(*)	onOffButton	setOff	n2	
n4	Mode	n6,n7				
n6	Cooler (*)		modeButton	setHeater	n7	
n7	Heater		modeButton	setCooler	n6	
n8	Speed	n10,n11,n12				
n10	Low (*)		speedButton	setMedium	n11	
n11	Medium		speedButton	setHigh	n12	
n12	High		speedButton	setLow	n10	
n13	Direction	n15				
n15	ShowDirection (*)					dirButton(n) /setDir(n)

Figure 4.7: Table created by O-Code for the state diagram of the Controller

in the state diagram become methods in the Controller class. A user has to insert body code for these methods. The main method is also declared in the Controller class.

2. To provide a common interface to all state classes, an abstract class, C_State, is defined. It contains empty declarations of operations for all events in the state diagram. Each state class has implementation code for its own events (operations). States in a state diagram may have *entry* and/or *exit* actions, 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. C_State provides empty declarations for entry and exit operations.

3. 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. If the state is a substate of another state then the class becomes a subclass of the superstate class. Otherwise, it is subclassed from the C_State class. If the state has *entry/exit* actions, methods having the names entry and exit, respectively,

```
class Controller {
  public static C_State state;
  public void onOffBut(){state.onOffBut();}
  public void modeBut(){state.modeBut();}
  ......
  public static void setOff(){...}
  public static void setCooler(){...}
  ......}

class C_State {/* Empty declarations for entry(), exit() and all methods of the subclasses of C_State*/}

class C_Operating extends C_State{
   static C_Mode mode; static C_Speed speed; static C_Direction direction;
   void entry() {Controller.setCooler();mode=new C_Cooler(); Controller.setLow();
     speed=new C_Low();Controller.setDir();direction=new C_ShowDirection();}
   void exit(){mode.exit();speed.exit();direction.exit();}
   void onOffBut() {exit();Controller.setOff();Controller.state=new C_Off();Controller.state.entry();}
   void modeBut() {mode.modeBut();}
   ......}

class C_Mode {/* Empty declarations for entry(), exit() and all methods of the subclasses of C_Mode*/}

class C_Heater extends C_Mode {
   void modeBut() {exit();Controller.setCooler();C_Operating.mode = new C_Cooler();
     C_Operating.mode.entry();}}

class C_ShowDirection extends C_Direction {
   void dirBut(int n) {Controller.setDir(n);}}
.......
```

Figure 4.8: Part of the generated code for the air conditioner

are defined in the class. Bodies of these methods contain a method-call to the corresponding *entry/exit* actions. All the state classes are defined in the same manner except the following two types which are defined differently:

(a) If the state is a superstate of AND-substates (e.g., Operating in Fig. 4.3), the class becomes a composite that contains as many objects as the number of the substates. For each substate, an attribute is defined. The name and type of the attribute are derived from the name of the substate. An entry method is defined which creates objects representing the default states within each of the concurrent substates and initializes each of the attributes with these objects (e.g., mode is initialized with an instance of C_Cooler class). Also, an exit method is defined which contains a call to the exit method of each of the attributes. 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).

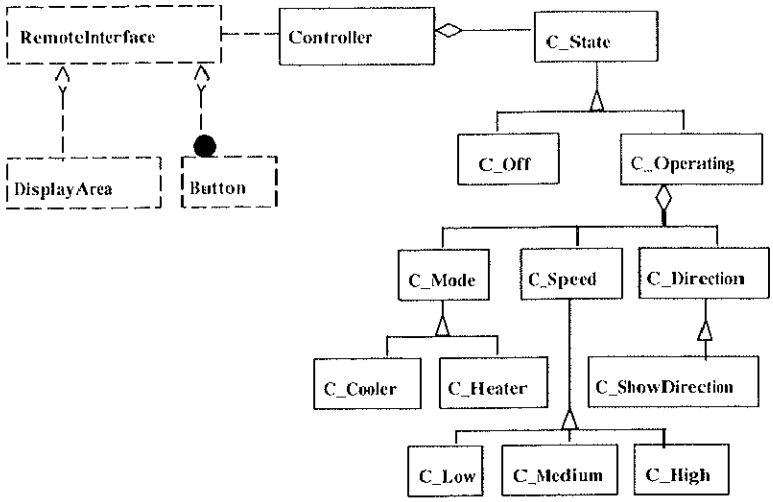


Figure 4.9: Class structure of the generated code for the air conditioner

The point to be noted here is that the system supports *synchronized* events in AND-substates. If there are transitions in many AND-substates for a single event, there will be implementation code in each of the substate classes, each of which will be called from the single method in the superstate class for the event.

(b) If the state is an AND-substate (e.g., Mode, Speed and Direction in Fig. 4.3), the class becomes an abstract class and serves as an interface for its own substate classes. In addition to the entry and exit operations, it contains empty declarations for operations corresponding to the events of its substates.

4. 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* one, 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) code for deleting the current state object and creating an object of the new state class, and (iii) call to the entry operation of the new state.

4.5 Executing the Generated Code

Figure 4.9 shows the class structure of the entire application for the air conditioner. Here, the solid-line boxes represent the classes that are generated by O-Code from the dynamic model of the air conditioner and the dashed-line boxes represent the domain classes, which came from the original object model. The clients of the Controller class do not need to know about the classes that implement the state-specific behavior of the Controller. They simply send their requests to the Controller object. The Controller object then forwards the same requests to its state object, which entertains them.

The Controller class has also an object of RemoteInterface class that is instantiated in its main method. While instantiating the RemoteInterface object, the Controller object passes itself as an argument so that the RemoteInterface object can later send requests to the Controller object whenever some button is pressed by the user. These requests from the RemoteInterface object become events for the Controller.

After compiling the code, when the Controller class is executed from the command line, the main method gets executed, and a RemoteInterface object is created. The RemoteInterface object displays itself and the interface shown in Figure 4.1 appears. Now when the user clicks some button, a message is sent to the Controller object. For example, when Mode button is clicked, the modeBut message will be sent. The Controller will send the same message to its state object (Figure 4.10). If the current state object contains an instance of C_Off class [case (a)], the empty modeBut method in the C_State class will be executed because there is no implementation for the modeBut operation in C_Off class. If, on the other hand, the current state object contains an instance of the C_Operating class and also the mode object of the the C_Operating contains an instance of the C_Heater class [case (b)], the modeBut method of the C_Operating class will get executed. This method sends the same message to the mode object (an instance of C_Heater class in this case). The modeBut method of the C_Heater class will get executed. This method first calls the setCooler method of the Controller class

and then update the mode object with a new instance of the C_Cooler class. The setCooler method of the Controller class, which corresponds to the *setCooler* action in the state diagram of the Controller, sends a setMode("COOLER") to the RemoteInterface object.

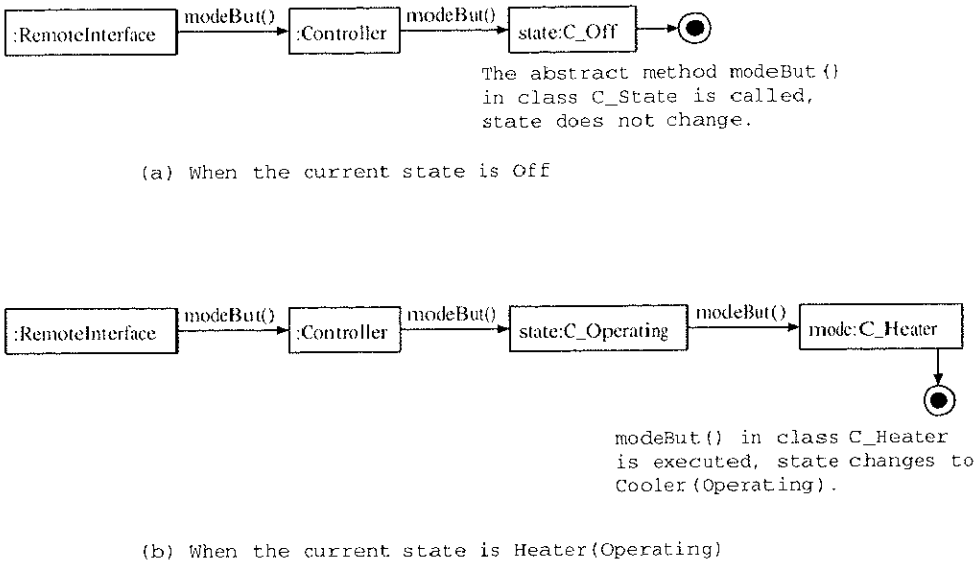


Figure 4.10: Sequence of operations when the Mode button is pressed

To summarize, the clients of the Controller class, e.g., the RemoteInterface object, send messages to the Controller that become events for it. The state object of the Controller, representing its current state, handles all the occurring events. While the events being handled, the state of the Controller may change and some method of the Controller class that represent actions in the state diagram may get executed. The action methods usually send messages to the clients of the Controller.

Chapter 5

Dynamic Model with Multiple State Diagrams and Activity Charts

5.1 Introduction

In the last two chapters, we described our approach regarding code generation from the dynamic model of small systems which can be represented by a single state diagram. The dynamic model of a real system, however, consists of multiple state diagrams, each of which shows the behavior of a particular class of objects. In this chapter, we demonstrate the code generation from the object and dynamic models having multiple state diagrams [17, 18]. We realized that the behavior of active objects, which keep their own control, can well be represented by activity diagrams [37].

5.2 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 Fourth International Workshop on Software Specification and Design [38]. 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 corresponding floor is visited by the elevator.

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.

5.3 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 5.1 shows the object diagram. We used the OMT notation except that the arrow headed lines between classes show possible messages between them.

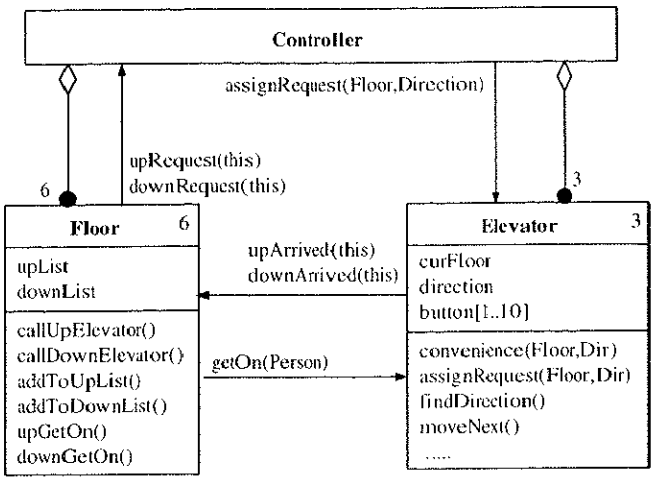


Figure 5.1: Object model for the elevator simulation system

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 object 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 elevator 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.

5.4 The **Controller** Class

Controller is a special class that keeps the main flow of control of a system [14, 15, 20, 21] 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. 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 the 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.

5.5 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. The behavior of a multi-state object is usually represented by a state diagram. 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 [33, 34, 35], which can contain OR-type or AND-type state hierarchy. Figure 5.2 shows the state diagram for the Floor class.

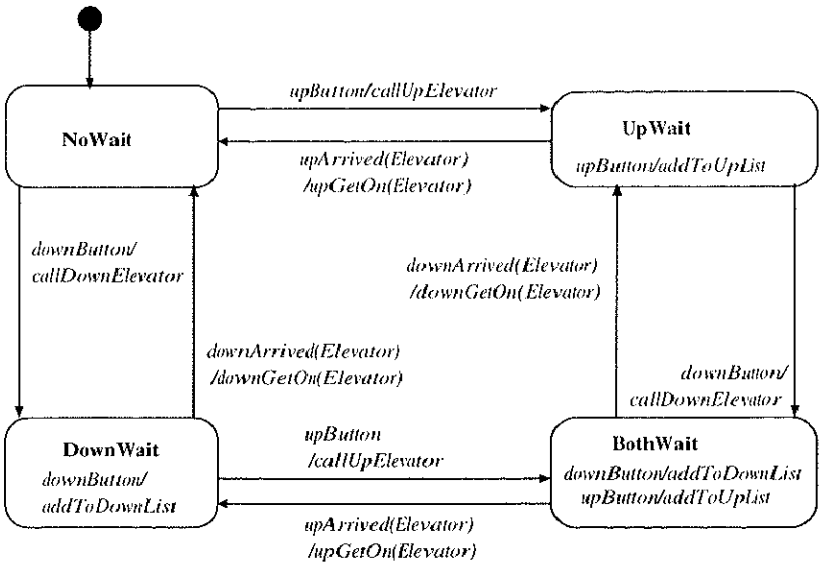


Figure 5.2: State diagram for Floor class

5.5.1 Converting a State Diagram into Code

To implement a state diagram, an object-oriented approach, described in the previous chapters, is used where each state becomes a class and each transition becomes an operation in that class. OR-type substates of a superstate becomes subclasses of the class that corresponds to the superstate. 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.

As can be seen in Figure 5.3, 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 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.

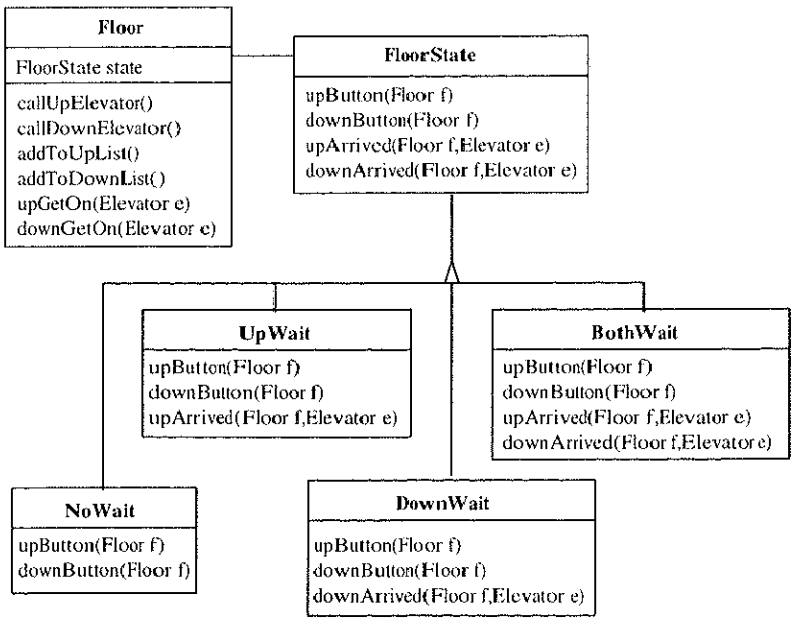


Figure 5.3: Class structure of the Floor class and its companion classes

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 its 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 {
   FloorState state = new NoWait(); //default state
   //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 of all the methods of its subclasses}
class NoWait extends FloorState {
   upButton(Floor f) {f.callUpElevator();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();}}
.....
```

5.5.2 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 {
   // create instances of the subclasses of FloorState beforehand
   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();

   FloorState state = NO_WAIT; //default state
   .....
}
class FloorNoWait extends FloorState {
   // use the already created instances instead of creating new ones
   upButton(Floor f) {f.callUpElevator();f.state = UP_WAIT;}
   downButton(Floor f) {f.callDownElevator();f.state = DOWN_WAIT;}
}
```

5.6 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 [37]. 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 sent it messages, which become events to trigger the transitions. Figure 5.4 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

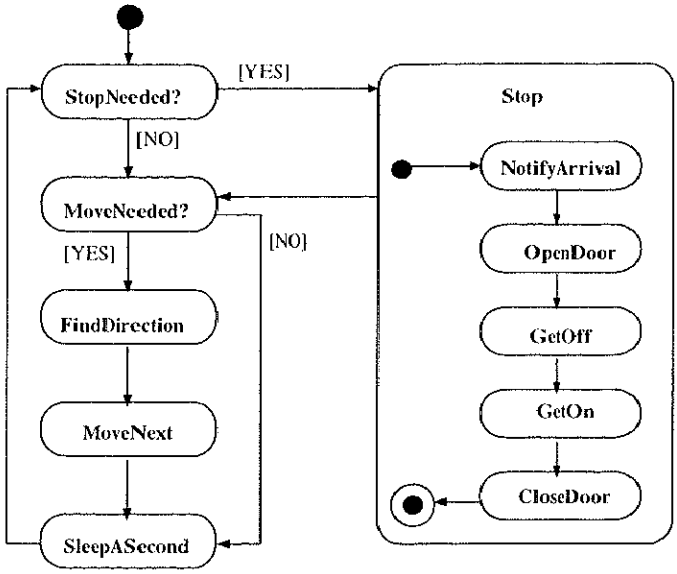


Figure 5.4: Activity diagram for Elevator class

the super-activity (in this case the *Stop* activity). Following is the implementation code generated from the activity diagram of the Elevator.

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

}

5.7 Classes having both State and Activity Diagrams

Classes of objects may have more than one aspects 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 5.4). 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, a *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(); //default elevator state
   //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 of all the methods of its 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();}}
```

5.8 Automatic Code Generation

The proposed approach has been implemented in our system, O-Code, which automatically generates executable Java code from the specifications of the object and dynamic models of a system. O-Code takes as input specifications of the object diagram, state diagrams and activity diagrams in Design Schema List (DSL) language [10]. O-Code generates Java code for different classes in the

following way.

5.8.1 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.

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. O-Code 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); // initializing permanent objects
    for (int i=0;i<elevatorList.length;i++){
      elevatorList[i] = new Elevator(i,this);// initializing permanent objects
      new Thread(elevatorList[i]).start();// creating threads for active objects
    }
  }
  public static void main(String args[]){
    Controller c = new Controller(); // instantiating Controller
```

```
    .....
  }
}
```

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 explained earlier. For each activity in the activity diagram, a method is declared in the class. Body code for these methods is entered by the user.

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. Body code for the action methods is entered by the user.

5.8.2 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.

   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.

   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.

Chapter 6

Automatic Code Generating System: O-Code

6.1 Introduction

In the previous chapters, we have mentioned several times our code generation system, O-Code, which generates Java code from the specifications of the dynamic model of a system. In this chapter, we describe the system in detail.

O-Code is developed in Java and is basically composed of three modules: **Transformer**, **Optimizer** and **Code Generator**. The source code is approximately 450, 200 and 400 lines respectively. The source code of the main module, which controls the above three modules, is approximately 350 lines.

Figure 6.1 shows the overall structure of the system. Modules are enclosed in round-corner rectangles and methods are enclosed in simple rectangles. A double line rectangle shows that the method is called several time from inside a loop. The flow of control goes from left to right inside modules. First the main module calls the Transformer module which reads the specifications of the dynamic model and converts the information into an intermediate (table) form. Then the main module calls the Optimizer module which reads the information in

the intermediate form and performs some optimization so that code can easily be generated from them. Finally the main module calls the Code Generator module which generates Java code for the application. The following sections give a brief description of the functionality of these three modules.

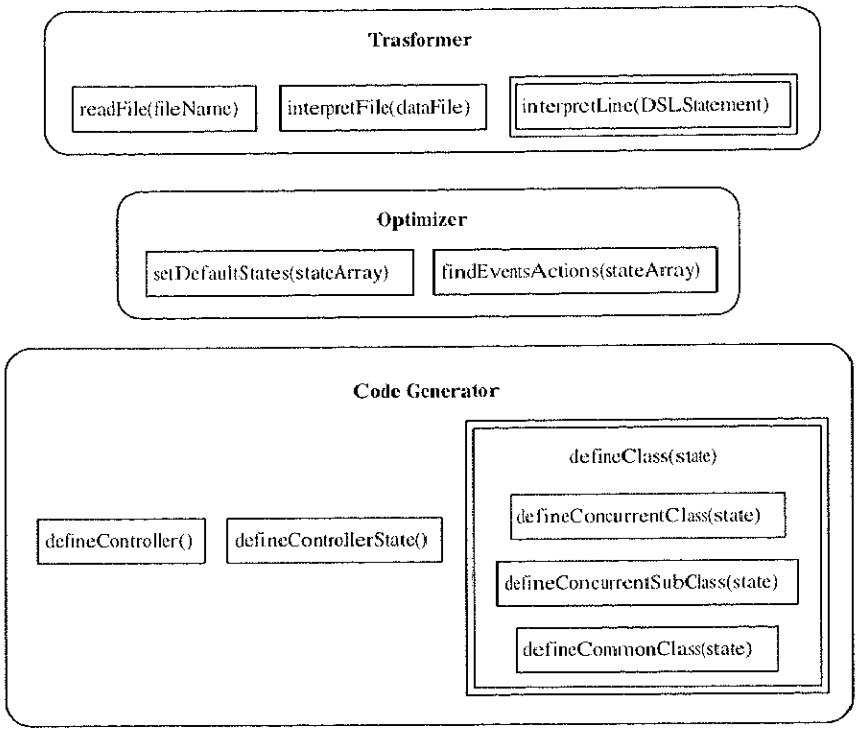


Figure 6.1: Overall structure of O-Code system

6.2 Transformer

The Transformer module reads the specifications of the state diagram, given in DSL format, and makes a table of states to properly record all the information, thus transforming the information from DSL format to a table format.

A number of classes have been used to form the structure of a nested table and to represent the elements of a state diagram. These classes include: State, Transition, Action, Event, Argument, Condition and InternalEvent. Figure 6.2 shows links between these classes. The figure shows that a State object can have a number

of Transition objects and/or InternalEvent objects associated with it. A Transition object can have an optional Action object, Event object and/or Condition object associated with it. An InternalEvent object always contains one Action object and one Event object. An Event object may have a number of Argument objects associated with it. The figure also shows that all objects except the State objects are maintained directly or indirectly by the State objects. The State objects themselves are maintained in a global array (stateArray) which is accessible to all modules.

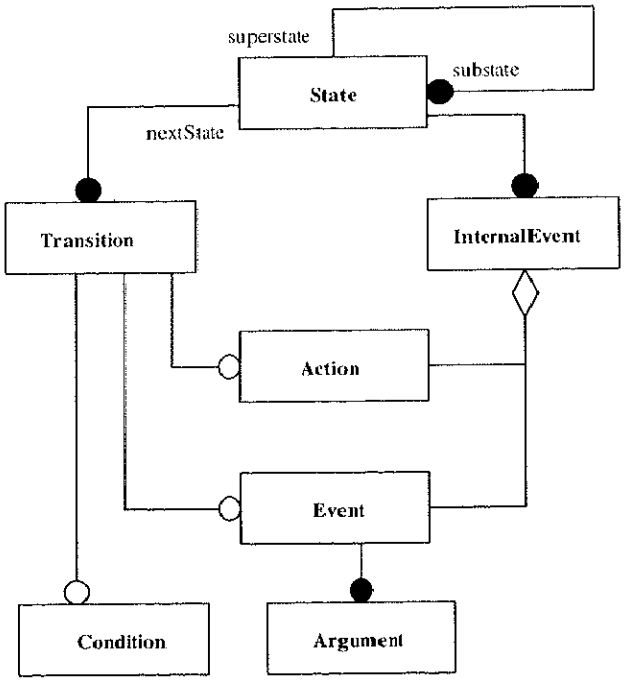


Figure 6.2: Classes that represents the elements of a state diagram

The Transformer module has a number of methods. The most important are readFile, interpretFile and interpretLine methods. Following is a brief description of the functionality of these methods.

6.2.1 The readFile Method

This method takes the name of the input file, which is in DSL format, as an input argument. It reads the file character by character, throws all white spaces and

creates a long string that contains all the DSL statements. It gives the long string (called dataFile) as the result. The dataFile is a global variable of type String.

6.2.2 The interpretFile Method

This method first splits the long dataFile string into several small strings each representing a DSL statement. A DSL statement always ends on a semicolon, so the dataFile string is split on semicolons. Each string, which represents a DSL statement, becomes an element of an array. The interpretFile method then starts a loop which calls the interpretLine method (explained below) for each string and passes the string as argument.

6.2.3 The interpretLine Method

This is quite a long method which takes a string representing a DSL statement as argument and interprets it. It collects the information contained in the DSL statement and, based on this information, instantiates objects of some of the classes shown in Figure 6.2. Sometimes it only initializes pointers in objects that are already created. For example, after reading the following DSL statement

```
OSTDN(n3)[loc(10:10),size(100:80),ostdnAttr(name:Operating,
concurrent{n4,n8,n13})];
```

a State object having the value of its Id attribute as "n3" will be searched. If the object does not exist, it will be created. The name and concurrent attributes of this object will be initialized with the values "Operating" and "true" respectively. The substates attribute which is an array of pointers to other State objects and represents the substates of the current state will contain pointers to the State objects having Ids "n4", "n8" and "n13".

6.3 Optimizer

After the Transformer module does its job, the information contained in the DSL file is converted into an intermediate form in which state diagram elements are represented as object instances. This information, however, is unorganized and needs to be optimized. For example, in a state diagram, the default state is indicated by an arc to the state from a small solid circle. This small circle merely shows that the adjacent node is a default one. But DSL, being graphical oriented, treats it as a node like any other node. Therefore, the Transformer module supposes it a state (having the name START). We need to ignore all such states and to consider the adjacent ones as the default states. Also, for code generation, we need to know not only the events that are supposed to occur on a state itself but also the events that may occur on its substates. The purpose of the Optimizer module is to refine the information given by the Transformer module in a way so that code can easily be generated from it. It also sorts out the table of states into a sequence so that super-states should always come before their substates. The setDefaultStates and findEventsActions are the important methods of the Optimizer module. Following is a brief description of these methods.

6.3.1 The setDefaultStates Method

This method follows the transitions from the states having their names as "START" and finds out the default states. It also eliminates all the State objects that have the name "START" from stateArray.

6.3.2 The findEventsActions Method

This methods finds out for each State object the events that occur on the substates of that state. It uses the pointers contained in the substates array and then follows the transitions and internalEvents of each of the substates to fetch the events and actions.

6.4 Code Generator

This module uses information given by the Optimizer module and generates Java language code. All the generated code is first written to a string buffer and in the end the buffer is written to disk.

The Code Generator module first executes the defineController and defineControllerState methods which generates code for the Controller and ControllerState classes respectively. A loop is then started which calls the defineClass method for each State object present in the stateArray and passes the state as the input argument. The defineClass method first finds the superstate of the state so that the class can be subclassed from the class for the superstate and then it calls one of the following four methods depending on the type of the state:

Concurrent State If the state has substates of AND-type, the defineConcurrentClass method is called.

Concurrent Sub-State If the state is an AND-type substate of another state, the defineConcurrentSubClass method is called.

Common State In all other cases, the defineCommonClass method is called.

The above three methods also call the defineTransition and defineInternalEvent methods when there is a transition and internal event on the state.

Chapter 7

Discussion

7.1 Discussion

We put all behavior associated with a particular state into one object. Because all state-specific code is contained in a single ControllerState subclass, new states and transitions can be added easily by defining new subclasses and operations. An alternative would be to use data values to define internal states and have Controller operations check the data explicitly. In such case, state transitions are implemented as assignments to some variables and have no explicit representation. As an example, Figure 7.1 shows code for the speedBut() method for the state diagram of the Air Conditioner (Figure 4.3). Note that transitions are implemented by assigning values to variables that represent various states. This may be good for efficiency but the actual behavior of the system that was represented as various states and transitions at the design stage, is buried into the code. It is very difficult to understand the behavior of the system by looking only at the code. Representing different states as separate objects makes the transitions more explicit and the code more understandable. This keeps the flavor of the state diagram in the implementation code and is also very helpful for reverse engineering the code back into the dynamic model.

Our approach may look like introducing too many classes, because the be-

```
class Controller {
  int mainState = 1; // 1=Off, 2=Operating
  int modeState = 1; // 1=Cooler, 2=Heater
  int speedState = 1; // 1=Low, 2=Medium, 3=High
  int directionState = 1; // 1=ShowDirection

  public void speedBut(){
    switch (mainState){
      case 1:{
        ...
        break;}
      case 2:
      switch (speedState){
        case 1:{
          speedState = 2;
          setMedium();
          break;}
        case 2:{
          speedState = 3;
          setHigh();
          break;}
        case 3:{
          speedState = 1;
          setLow();
          break;}
      }
    }
  }

  ...
}
```

Figure 7.1: Implementing a state diagram using data values to represent states

havior for different states is distributed across several ControllerState subclasses. This increases the number of classes and the implementation of behavior is less compact than a single class. However, such distribution eliminates large conditional statements. Large conditional statements are undesirable because they tend to make the code less understandable and difficult to modify and extend. Encapsulating each state transitions and actions in a class elevates the idea of an execution state to full object status. Furthermore, for the sake of automatic code generation, it is very natural to have a one-to-one correspondence between the components of state diagrams (such as states and transitions) and the program elements (such as classes and methods). That is why, well-known code generating tools, like Rhapsody (i-Logix) [31, 32], also represent states as classes.

In our approach, a subclass of ControllerState is instantiated whenever a transition is executed. It seems that the resulting code is less efficient. A solution to this problem is to create objects of all subclasses of ControllerState ahead of

time and then, while executing a transition, assign an appropriate instance to the state object instead of instantiating a new one. This has been shown in Chapter 5 under [Optimizing the Code].

7.2 Comparison with Rhapsody

Rhapsody (i-Logix) [32], which is a successor of O-Mate [31], is a tool that allows to create object diagram, state transition diagrams and message sequence charts for an application and then automatically generates C++ code for the application. Because Rhapsody is the only tool of its kind, we compare the code generated by O-Code to that of Rhapsody.

The similarity between O-Code and Rhapsody is that both of the systems treat the states of a state diagram as objects. The differences lie in the treatment of state hierarchy, events and transitions. The details of converting a state diagram into code are not fully given in the papers [31, 32], where Rhapsody is reported. But looking at the code generated by it, one can understand the structure of the code. As shown in Table 7.1, in Rhapsody events are implemented as classes; state hierarchy is implemented by having pointers to super/sub-state classes; and transition searching is performed by executing a *switch* statement. Whereas in O-Code, events become methods; state hierarchy is implemented by inheritance; and transition searching is automatically performed by using the concept of polymorphism. As shown below, these differences give an edge to O-Code over Rhapsody in terms of compactness, efficiency and readability of the generated code.

7.2.1 Executing Image of the Code Generated by Rhapsody

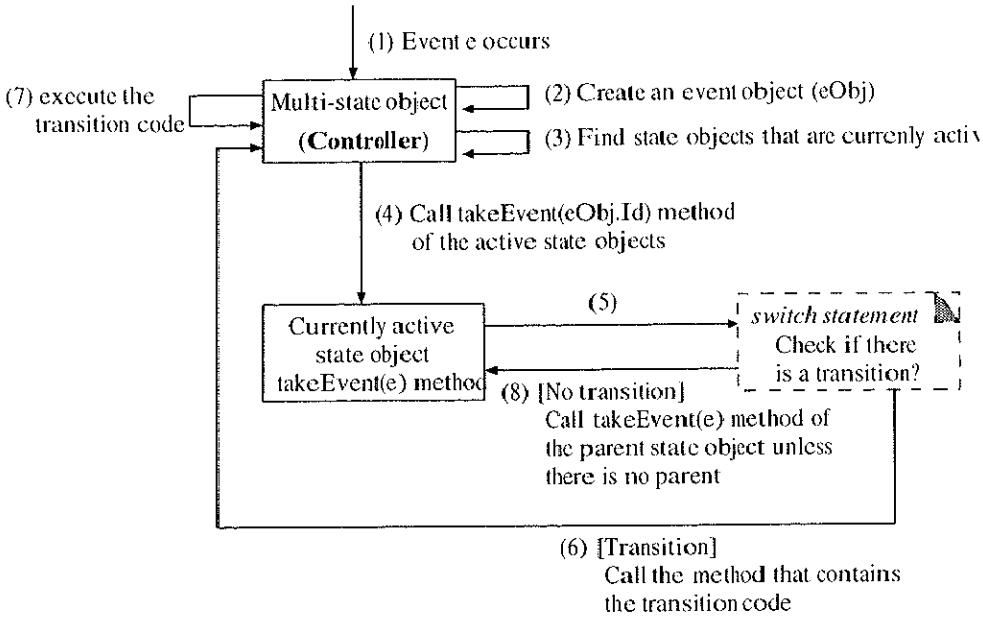
In Rhapsody, all states and events become classes. There is an abstract class State from which four classes: AndState, ComponentState, OrState and LeafState

State diagram components	Rhapsody's implementation	O-Code's implementation
States	Become classes	Become classes
Events	Become classes	Become methods
State hierarchy	Using pointers	Using inheritance
Transition searching	Using *switch* statement	Using Polymorphism

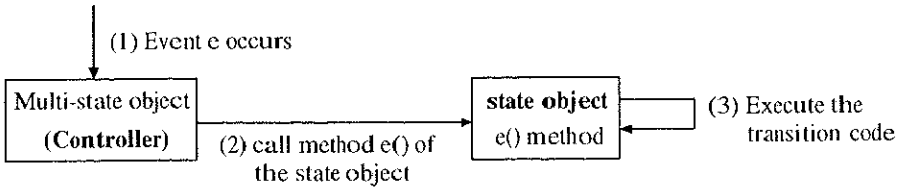
Table 7.1: Comparing the mechanisms used by Rhapsody and O-Code to implement a state diagram

are derived. The four classes implement respectively the general behavior of four types of states: superstate of AND-substates, AND-substate, superstate of OR-substates 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. One object instance is created for each state, and the Controller class keeps references to all these objects. The Controller object can always find which state is active, because all the state classes have 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 Controller class. As shown in Figure 7.2, When a client of the Controller sends a request to it, this becomes an event and a corresponding event object is created (Steps 1 and 2). The Controller object calls the takeEvent(EventId) method of the active state object and passes the event object as an argument (Steps 3 and 4). This method contains a *switch* statement that finds if there is any transition on this event from the current state (Step 5). If there is a transition, the corresponding method in the Controller class is made executed (Steps 6 and 7), which updates the current state of the Controller. If there is no transition, the event is sent to the object representing the parent state (Step 8). All this when sum up takes a considerable amount of time, in spite of the fact that all state objects are created ahead of time and not during transitions.



(a) Executing image of the code generated by Rhapsody



(b) Executing image of the code generated by O-Code

Figure 7.2: A conceptual view of the execution sequence in code generated by Rhapsody and O-Code

7.2.2 Executing Image of the Code Generated by O-Code

In O-Code, states become classes but events become methods. Since state hierarchy is implemented by inheritance, substates do not need to have pointers to their superstates. For each transition there is a method defined in the class corresponding to the state from which the transition is originated.

In our code, when an event occurs on which there is a transition, the method defined for that transition in the current state class is executed. When an event having no transition occurs, there will be no method for it in the current state class and therefore an abstract method of the ControllerState will be called, which is a fast operation. Nothing more happens. That is why, the time taken by our code for such events is markedly short. If the event has a transition, the method in the concrete state class gets executed which causes creating a new state object and thus takes some time. However, as there are no conditional structures in the code, the time is still shorter than that of Rhapsody's code.

7.2.3 Comparing the Code Generated by Rhapsody and O-Code

We used the same air conditioner example (Chapter 4) and compared the code generated by Rhapsody to that of O-Code. To have a fair comparison, we rewrote the code generated by Rhapsody in Java, because O-Code generates Java code whereas Rhapsody generates C++ code. Findings of the comparison are as follow;

1. *Code generated by O-Code is more compact.* The original C++ code that is 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 O-Code, as shown in Table 7.2. 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.

	Rhapsody	O-Code
Source code: No. of lines	982	219
Source code: No. of bytes	19458	4413
No. of classes	23	13

Table 7.2: Comparing the compactness of the code generated by Rhapsody and O-Code

2. *Our code is more efficient than Rhapsody's code.* To compare the efficiency of the code generated by O-Code and Rhapsody, we performed an experiment in which the same sequence of 1000 requests was sent to the Controller class. Out of these 1000 events, 556 caused transitions while the remaining 444 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 7.3, 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 O-Code offers for all types of events is 38.00%.

3. *Rhapsody code is less understandable.*

   As explained above, though Rhapsody implements state-specific behavior in separate classes, it puts the transition-selection code in the *case* structure inside the takeEvent(EventId) method of the state classes. Actual transitions are implemented as methods in the Controller class, which are called when the current state object succeeds in finding a transition on an event. This makes the code difficult to understand.

	Rhapsody (x) (millisecs.)	O-Code (y) (millisecs.)	Improvement $(x-y)/x*100$
Total time for events without transitions (a)	127.8000	54.3000	
Average time per event without transition ($a/444$)	0.2299	0.0977	57.50%
Total time for events having transitions (b)	144.7500	114.6500	
Average time per event having transition ($b/556$)	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 7.3: Comparing the efficiency of the code generated by Rhapsody and O-Code

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 case structures. This contributes to making the code more understandable.

Chapter 8

Related Work

8.1 Code Generating CASE Tools

The most related work is that of Harel and Gery [31, 32] whose tool, Rhapsody [39], generates C++ code from the object and dynamic models. As described earlier, 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 [37] 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 [40], ROOM [41], Graphical Designer [42] and StP [43], can generate code from the state diagrams, but they do not usually support state hierarchy and concurrent states in the state diagrams.

SoftReuse [44, 45] extracts program specifications from domain models and

automatically generates programs. The domain models are described in the program specification description language (PSDL) [46] which is based on ER model. Before extracting program specifications, constraints about attribute values and input-output data are defined. Looking from OMT, SoftReuse only considers the object and functional models and does not consider the dynamic model. Therefore, it cannot be applied to applications having dynamic behavior or control.

8.2 Implementing State Diagrams

In the traditional approach [29], a finite state machine is implemented by representing it as a table and writing an interpreter to execute the table. This is a straightforward but less efficient approach. In addition, implementing state hierarchy and concurrent states is quite difficult.

Rumbaugh [20] has described an approach in which he has used inheritance to implement state hierarchy but he does not consider concurrency and active objects.

Our mechanism of converting a state diagram into implementation code has some similarity with the State pattern [47], but State pattern neither addresses the issue of state hierarchy nor does it address concurrency within state diagrams.

The relation between states and classes is examined by Ran [48]. Sane and Campbell [49] say that states can be represented as classes and transitions as operations. They implement embedded states by making a table for the superstate and do not consider concurrent states. An object-oriented extension to the state diagram has been done by Coleman et al. [50].

8.3 Other

Harada et al. [51, 52] have developed tools which convert the analysis and design models of Structured Object Modeling Method (SOMM) into Design Schema List (DSL) language [10]. The tools also includes a reverse engineering system OORE, which generates design elements from a given C++ program.

Nakashima et al. [53, 12, 13] have developed tools which provide graphical editors to draw various OMT diagrams interactively; compute layouts for the diagrams using drawing layout algorithms; and generate DSL representation of the diagrams.

Joung et al. [19] show how icons can be added to the state diagrams and then code can be generated that animates the system.

Chapter 9

Conclusions

An object oriented approach has been proposed to convert the dynamic model of a system, represented as a set of state transition diagrams and activity diagrams, into implementation code. State transition diagrams are used to show the behavior of multi-state objects and activity diagrams are used to represent the behavior of active objects which keep their own threads of control.

Using our approach, states in the state diagrams are represented as classes and transitions as operations eliminating the need of using large conditional statements. This makes the components of the state diagram explicit and the resulting code easier to understand. 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 are implemented as Java threads. The method deals with intra-object concurrency (within a single object) and multiple thread concurrency (among several objects).

The proposed method has been implemented in our system, O-Code, which automatically converts the dynamic model specifications into executable Java code. The comparison with Rhapsody shows that the code generated by our system is approximately 40% more efficient and five times more compact than that of Rhapsody.

Acknowledgments

I wish to express my sincere thanks and profound gratitude to my supervisor Dr. Jiro Tanaka, Associate Professor, University of Tsukuba, for his invaluable guidance, advice, supervision and constant encouragement during the course of the present study. The study would not have been possible without his generous training.

I am highly obliged to Dr. Kozo Itano, Dr. Nobuo Ohbo, Dr. Seiichi Nishihara, and Dr. Kazuhiko Kato of University of Tsukuba for serving as member of the examination committee and critical readings of the thesis and for their specific suggestions to improve the manuscript.

I thank members of HITO project: Dr. Minoru Harada, Associate Professor Aoyama Gakuin University; Dr. Kiyoshi Itoh, Professor Sophia University; and Dr. Atsushi Ohnishi, Professor Ritsumeikan University, for their advice and comments.

I am grateful to all the reviewers who critically read my papers and their comments improved the quality of this work.

Special thanks to Mr. Sucktae Joung, Mr. Satoshi Nakashima, Mr. Tohru Ogawa, and all other members of IP-Lab, University of Tsukuba, for useful discussions, constructive criticism and timely help.

I wish to express my thanks to all my friends in Pakistan and Japan for their encouragement during this work.

Deep appreciations to my parents, brothers and sisters for their benevolent prayers. Last but not least to my family Nighat, Suhani and Uzair for their heartfelt support, love and prayers during their stay in Pakistan and Japan.

Bibliography

[1] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Eaglewood Cliffs, New Jersey, 1991.

[2] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, Redwood, California, 1991.

[3] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, Eaglewood Cliffs, New Jersey, 1991.

[4] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley, Reading, Massachusetts, 1992.

[5] Philippe Desfray. *Object Engineering: The Fourth Dimension.* Addison Wesley, Reading, Massachusetts, 1994.

[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] Taegyun Kim. Object oriented designer. http://www.qucis.queensu.ca/Software-Engineering/blurb/OOD.html, ftp.x.org.

[10] Minoru Harada, Terutada Fujisawa, Masataka Teradaira, Kouji Yamamoto, and Susumu 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 '96*, pages 111–118, Tokyo, Japan, 1996. IPSJ. (in Japanese).

[11] M Ohno and M Harada. Computer automated modeling engine for objects – automatic extraction and classification of design elements from texts. *Transactions of Information Processing Society, Japan (94-SE-99)*, pages 105–112, 1994. (in Japanese).

[12] Satoshi Nakashima, Jauhar Ali, and Jiro Tanaka. Applying graph drawing algorithm to omt diagram. In *Proceedings of International Symposium on Future Software Technology (ISFST-96)*, pages 18–25, Xi'an, China, October 1996.

[13] Satoshi Nakashima, Jauhar Ali, and Jiro 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. SDPS.

[14] Jauhar Ali and Jiro 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. SDPS.

[15] Jauhar Ali and Jiro 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. IASTED.

[16] Jauhar Ali and Jiro Tanaka. An object oriented approach to generate executable code from the omt-based dynamic model. *Journal of Integrated Design and Process Science*. (to appear).

[17] Jauhar Ali and Jiro Tanaka. Implementation of the dynamic behavior of object oriented system. In *Proceedings of the Third World Conference on In-*

tegrated Design and Process Technology, Berlin, Germany, July 1998. SDPS. (to appear).

[18] Jauhar Ali and Jiro Tanaka. Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. In *Proceedings of AoM/IAoM 16th Annual International Conference*, Chicago, USA, August 1998. SDPS. (to appear).

[19] Sucktae Joung, Jauhar Ali, and Jiro 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. Software Engineers Association, Japan.

[20] James Rumbaugh. Controlling code: How to implement dynamic models. *Journal of Object-Oriented Programming*, 6(2):25–30, May 1993.

[21] James Rumbaugh. Objects in the twilight zone: How to find and use application objects. *Journal of Object-Oriented Programming*, 6(3):18–23, June 1993.

[22] James Rumbaugh. The life of an object model: How the object model changes during development. *Journal of Object-Oriented Programming*, pages 24–32, April 1994.

[23] James Rumbaugh. Going with the flow: Flow graphs in their various manifestations. *Journal of Object-Oriented Programming*, pages 12–23, June 1994.

[24] James Rumbaugh. Getting started: Using use cases to capture requirements. *Journal of Object-Oriented Programming*, pages 8–12, September 1994.

[25] James Rumbaugh. Omt: The object model. *Journal of Object-Oriented Programming*, 7(8):21–27, January 1995.

[26] James Rumbaugh. Omt: The dynamic model. *Journal of Object-Oriented Programming*, 7(9):6–12, February 1995.

[27] James Rumbaugh. Omt: The development process. *Journal of Object-Oriented Programming*, pages 8–16, May 1995.

[28] Grady Booch. *Object Solutions: Managing the object-oriented project.* Addison Wesley, 2725 Sand Hill Road Menlo Park, CA 94025, 1996.

[29] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Computer Design.* Addison Wesley, Reading, Massachusetts, 1979.

[30] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, Reading, Massachusetts, 1996.

[31] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of 18th International Conference on Software Engineering*, pages 246–257. IEEE, March 1996.

[32] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

[33] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, (8):231–274, August 1987.

[34] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[35] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[36] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Eaglewood Cliffs, New Jersey, 1988.

[37] Rational Software Corporation. *Unified Modeling Language (UML).* http://www.rational.com.

[38] IEEE Computer Society. *Problem Set for the Fourth International Workshop on Software Specification and Design*, April 1987.

[39] i-Logix Inc. *Rhapsody.* http://www.ilogix.com.

[40] MicroGold Software, NJ, 08807. *StateMaker*. Internet 71543.1172@compuserve.com, http://www.worldwidemart.com/mattw/software/Windows3.X/demo/.

[41] ObjectTime Limited. *ROOM*. http://www.objectime.on.ca/.

[42] Advanced Software Technologies. *Graphical Designer*. http://www.advancedsw.com/.

[43] Aonix. *StP: Software Trough Pictures*. http://www.ide.com/index.html.

[44] Masaaki Hashimoto, Toyohiko Hirota, and Kazuhisa Yokota. An experiment on reusing software based on domain model. *Transactions of Information Processing Society, Japan*, 36(5):1040–1049, 1995. (in Japanese).

[45] K Yokota, Masaaki Hashimoto, and M Sato. An experiment on reusing program specifications described with conceptual data model and dependency constraint-based language. In *Proceedings of International Conference on Computing and Information*, pages 324–328, 1992.

[46] Masaaki Hashimoto and K Okamoto. A set and mapping-based dectection and solution method for structure clash between program input and output data. In *Proceedings of Computer Software and Application Conference*, pages 629–638. IEEE, 1990.

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1995.

[48] Alexander S. Ran. Modeling states as classes. In *Proceedings of the Technology of Object-Oriented Languages and Systems Conference*, 1994.

[49] Aamod Sane and Roy Campbell. Object-oriented state machines: Subclassing, composition, delegation, and genericity. In *ACM SIGPLAN Notices, OOPSLA'95*, volume 30, pages 17–32, Austin, Texas, October 1995. ACM.

[50] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[51] Minora Harada, Takafumi Sawada, and Terutada Fujisawa. A structured object modeling method somm and its environment some. *Systems and Computers in Japan*, 27(11):1–18, 1996.

[52] Minoru Harada, Kazuhiro Kitamoto, and Takashi Iwata. The structure object modeling environment some which visualizes both the control structure and the event-sending. In *Object Oriented Symposium '97*, pages 136–144, TokyoJapan, 1997. IPSJ. (in Japanese).

[53] Satoshi Nakashima and Jiro Tanaka. An automatic layout system for omt-based object diagram. In *Object-Oriented Symposium '96*, pages 103–110, TokyoJapan, 1996. IPSJ. (in Japanese).

筑波大学附属図書館
1 00993 11258 9

本学関係