5

Dynamic Modeling

Temporal relationships are difficult to understand. A system can best be understood by first examining its static structure, that is, the structure of its objects and their relationships to each other at a single moment in time. Then we examine changes to the objects and their relationships over time. Those aspects of a system that are concerned with time and changes are the *dynamic model,* in contrast with the static, or object model. *Control* is that aspect of a system that describes the sequences of operations that occur in response to external stimuli, without consideration of what the operations do, what they operate on, or how they are implemented.

This chapter describes concepts dealing with flow of control, interactions, and sequencing of operations in a system of concurrently-active objects. The major dynamic modeling concepts are *events,* which represent external stimuli, and *states,* which represent values of objects. The *state diagram* is a standard computer science concept (a graphical representation of finite state machines) that has been handled in different ways in the literature, depending on its use. We emphasize the use of events and states to specify control, rather than as algebraic constructs. We show that states and events can be organized into generalization hierarchies to share structure and behavior.

In this chapter we mainly follow the notation of David Harel [Harel-87] for drawing structured state diagrams using nested contours to show structure.

5.1 EVENTS AND STATES

An object model describes the possible patterns of objects, attributes, and links that can exist in a system. The attribute values and links held by an object are called its *state*. Over time, the objects stimulate each other, resulting in a series of changes to their states. An individual stimulus from one object to another is an *event*. The response to an event depends on the state of the object receiving it, and can include a change of state or the sending of another event

to the original sender or to a third object. The pattern of events, states, and state transitions for a given class can be abstracted and represented as a *state diagram*. A state diagram is a network of states and events, just as an object diagram is a network of classes and relationships. The *dynamic model* consists of multiple state diagrams, one state diagram for each class with important dynamic behavior, and shows the pattern of activity for an entire system. Each state machine executes concurrently and can change state independently. The state diagrams for the various classes combine into a single dynamic model via shared events.

5.1.1 Events

An *event* is something that happens at a point in time, such as *user depresses left button* or *Flight 123 departs from Chicago*. An event has no duration. Of course, nothing is really instantaneous; an event is simply an occurrence that is fast compared to the granularity of the time scale of a given abstraction.

One event may logically precede or follow another, or the two events may be unrelated. Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are causally related. Flight 123 may depart before or after Flight 456 departs Rome; the two events are causally unrelated. Two events that are causally unrelated are said to be *concurrent;* they have no effect on each other. If the communications delay between two locations exceeds the difference in event times, then the events must be concurrent because they cannot influence each other. Even if the physical locations of two events are not distant, we consider the events concurrent if they do not affect each other. In modeling a system we do not try to establish an ordering between concurrent events because they can occur in any order. Any realistic model of a distributed system must include concurrent events and activities.

An event is a one-way transmission of information from one object to another. It is not like a subroutine call that returns a value. In the real world, all objects exist concurrently. An object sending an event to another object may expect a reply, but the reply is a separate event under the control of the second object, which may or may not choose to send it.

Every event is a unique occurrence, but we group them into *event classes* and give each event class a name to indicate common structure and behavior. This structure is hierarchical, just as object class structure is hierarchical. For example, *Flight 123 departs from Chicago* and *Flight 456 departs from Rome* are both instances of event class *airplane flight departs*. Some events are simple signals, but most event classes have attributes indicating the information they convey. For example, *airplane flight departs* has attributes *airline, flight number,* and *city*. The time at which an event occurs is an implicit attribute of all events.

An event conveys information from one object to another. Some classes of events may be simply signals that something has occurred, while other classes of events convey data values. The data values conveyed by an event are its *attributes,* like the data values held by objects. Attributes are shown in parentheses after the event class name. Figure 5.1 shows some examples of event classes with attributes. Showing attributes is optional.

The term *event* is often used ambiguously. Sometimes event refers to an event instance, at other times to an event class. In practice, this ambiguity is usually not a problem and the precise meaning is apparent from the context.

```
airplane flight departs (airline, flight number, city)
mouse button pushed (button, location)
input string entered (text)
phone receiver lifted
digit dialed (digit)
engine speed enters danger zone
```

Figure 5.1 Event classes and attributes

Events include error conditions as well as normal occurrences. For example, *motor jammed, transaction aborted,* and *time-out* are typical error events. There is nothing different about an error event; only our interpretation makes it an "error."

5.1.2 Scenarios and Event Traces

A *scenario* is a sequence of events that occurs during one particular execution of a system. The scope of a scenario can vary; it may include all events in the system, or it may include only those events impinging on or generated by certain objects in the system. A scenario can be the historical record of executing a system or a thought experiment of executing a proposed system.

Figure 5.2 shows a scenario for using a telephone line. This scenario only contains events affecting the phone line.

```
caller lifts receiver
dial tone begins
caller dials digit (5)
dial tone ends
caller dials digit (5)
caller dials digit (5)
caller dials digit (1)
caller dials digit (2)
caller dials digit (3)
caller dials digit (4)
called phone beings ringing
ringing tone appears in calling phone
called party answers
called phone stops ringing
ringing tone disappears in calling phone
phones are connected
called party hangs up
phones are disconnected
caller hangs up
```

Figure 5.2 Scenario for phone call

Each event transmits information from one object to another. For example, *dial tone begins* transmits a signal from the phone line to the caller. The next step after writing a scenario is to identify the sender and receiver objects of each event. The sequence of events and the objects exchanging events can both be shown in an augmented scenario called an *event trace*

diagram. This diagram shows each object as a vertical line and each event as a horizontal arrow from the sender object to the receiver object. Time increases from top to bottom, but the spacing is irrelevant; it is only the sequences of events that are shown, not their exact timing. (Real-time systems impose time constraints on event sequences, but that is a separate matter requiring extra notation.) Figure 5.3 shows an event trace for a phone call. Note that concurrent events can be sent (*Phone line* sends events to *Caller* and *Callee* concurrently) and events between objects need not alternate (*Caller* dials several digits in succession).

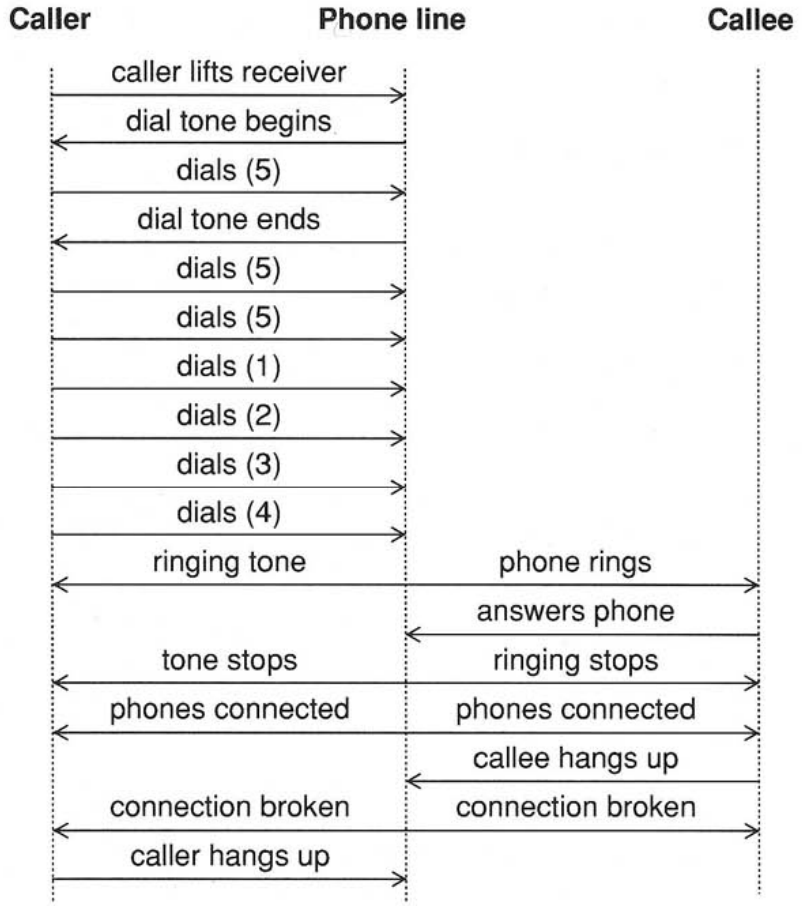

Figure 5.3 Event trace for phone call

5.1.3 States

A *state* is an abstraction of the attribute values and links of an object. Sets of values are grouped together into a state according to properties that affect the gross behavior of the object. For example, the state of a bank is either solvent or insolvent, depending on whether its assets exceed its liabilities. A state specifies the response of the object to input events. The response to an event received by an object may vary quantitatively depending on the exact values of its attributes, but the response is qualitatively the same for all values within the same state, and may be qualitatively different for values in different states. The response of

an object to an event may include an action or a change of state by the object. For example, if a digit is dialed in state *Dial tone,* the phone line drops the dial tone and enters state *Dialing*; if the receiver is replaced in state *Dial tone*, the phone line goes dead and enters state *Idle*.

A state corresponds to the interval between two events received by an object. Events represent points in time; states represent intervals of time. For example, after the receiver is lifted and before the first digit is dialed, the phone line is in state *Dial tone*. The state of an object depends on the past sequence of events it has received, but in most cases past events are eventually hidden by subsequent events. For example, events that happened before the phone is hung up have no effect on future behavior; the *Idle* state "forgets" events received prior to the *hang up* event.

A state has duration; it occupies an interval of time. A state is often associated with a continuous activity, such as the ringing of a telephone, or an activity that takes time to complete, such as flying from Chicago to San Francisco. Events and states are duals of one another; an event separates two states, and a state separates two events.

A state is often associated with the value of an object satisfying some condition. For example, *water is liquid* is equivalent to saying "the temperature of the water is greater than 0 C and less than 100 C." In the simplest case, each enumerated value of an attribute defines a separate state. For example, an automobile transmission might be in states *Reverse, Neutral, First, Second,* or *Third.*

In defining states, we ignore those attributes that do not affect the behavior of the object, and we lump together in a single state all combinations of attribute values and links that have the same response to events. Of course, every attribute has some effect on behavior or it would be meaningless, but often some attributes do not affect the pattern of control and can be thought of as simple parameter values within a given state. Recall that the purpose of modeling is to focus on those qualities of an entity that are relevant to the solution of an application problem and abstract away those that are irrelevant. The three different OMT models (object, dynamic, and functional) present different views of a system; the particular choice of attributes and values are not equally important in these three different views. For example, except for leading 0s and 1s, the exact digits dialed do not affect the control of the phone line, so we can summarize them all with state *Dialing* and track the phone number as a parameter. Sometimes, all possible values of an attribute are important but usually only when the number of possible values is small.

Both events and states depend on the level of abstraction used. For example, a travel agent planning an itinerary would treat each segment of a journey as a single event; a flight status board in an airport would distinguish departures and arrivals; an air traffic control system would break each flight into many geographical legs.

A state can be characterized in various ways. Figure 5.4 shows various characterizations of the state *Alarm ringing* on a watch. The state has a suggestive name and a natural-language description of its purpose. The event sequence that leads to the state consists of setting the alarm, doing anything that doesn't clear the alarm, and then having the target time occur. A declarative condition for the state is given in terms of parameters, such as *alarm* and *target time*; the alarm stops ringing after 20 seconds. Finally, a stimulus-response table shows the

State: *Alarm ringing*

Description: alarm on watch is ringing to indicate target time

Event sequence that produces the state:

set alarm (target time)

any sequence not including *clear alarm*

current time = target time

Condition that characterizes the state:

alarm = on, and target time ≤ current time ≤ target time + 20 seconds, and no button has not been pushed since target time

Events accepted in the state:

event	action	next state
current time = target time + 20	reset alarm	*normal*
button pushed (any button)	reset alarm	*normal*

Figure 5.4 Various characterizations of a state

effect of events *current time* and *button pushed*, including the action that occurs and the next state. The different descriptions of a state may overlap.

Can links have state? In as much as they can be considered objects, links can have state. As a practical matter, it is generally sufficient to associate state only with objects. The state of an object can include the values of its links.

5.1.4 State Diagrams

A *state diagram* relates events and states. When an event is received, the next state depends on the current state as well as the event; a change of state caused by an event is called a *transition*. A state diagram is a graph whose nodes are states and whose directed arcs are transitions labeled by event names. A state is drawn as a rounded box containing an optional name. A transition is drawn as an arrow from the receiving state to the target state; the label on the arrow is the name of the event causing the transition. All the transitions leaving a state must correspond to different events.

The state diagram specifies the state sequence caused by an event sequence. If an object is in a state and an event labeling one of its transitions occurs, the object enters the state on the target end of the transition. The transition is said to *fire*. If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire. If an event occurs that has no transition leaving the current state, then the event is ignored. A sequence of events corresponds to a path through the graph.

Figure 5.5 shows a state diagram describing the behavior of a telephone line. The diagram is drawn for one phone line, not the caller or callee. The diagram contains sequences associated with normal calls as well as some abnormal sequences, such as timing out while

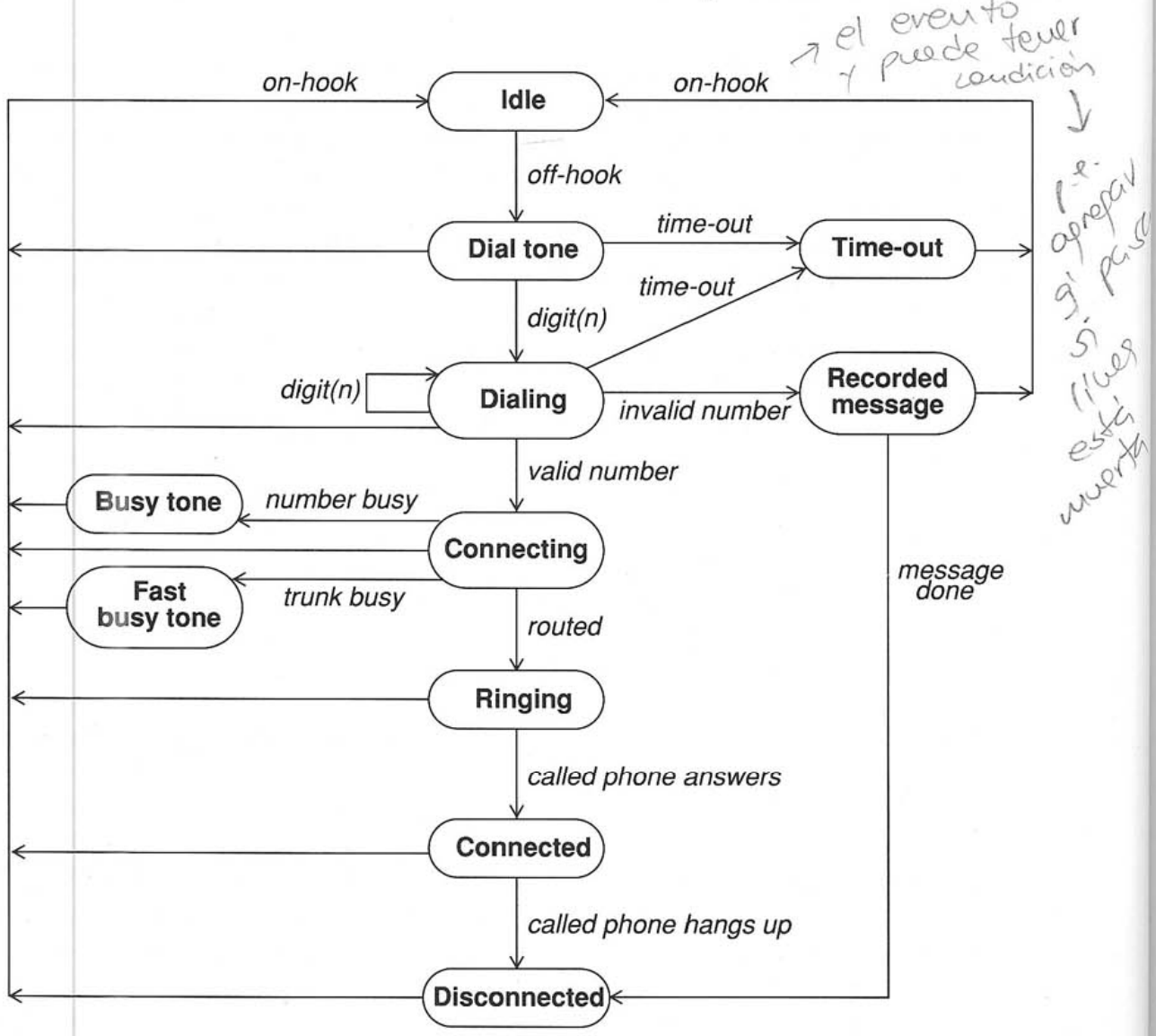

Figure 5.5 State diagram for phone line

dialing or getting busy lines. The event *on-hook* causes a transition from any state to the *Idle* state; this is drawn as a bundle of transitions leading to *Idle*. Later we will show a more general notation that represents events applicable to groups of states with a single transition.

Note that the states do not totally define all values of the object. For example, state *Dialing* includes all sequences of incomplete phone numbers. It is not necessary to distinguish between different numbers as separate states since they all have the same behavior, but the actual number dialed must of course be saved as an attribute.

A state diagram describes the behavior of a single class of objects. Since all instances of a class have the same behavior (by definition), they all share the same state diagram, as they all share the same class features. But as each object has its own attribute values, so too each object has its own state, the result of the unique sequence of events that it has received. Each object is independent of other objects and proceeds at its own pace.

State diagrams can represent one-shot life cycles or continuous loops. The diagram for the phone line is a continuous loop. In describing ordinary usage of the phone, we do not know or care how the loop is started. (If we were describing installation of new lines, the initial state would be important.) One-shot diagrams represent objects with finite lives. A one-shot diagram has initial and final states. The initial state is entered on creation of an object; entering the final state implies destruction of the object. An initial state is shown by a solid circle. The circle can be labeled to indicate different initial conditions. A final state is shown by a bull's-eye. The bull's-eye can be labeled to distinguish final conditions. Figure 5.6 shows the life cycle of a chess game (with some simplifications). A one-shot diagram can be considered a state diagram "subroutine" that can be referenced from various places in a high-level diagram. Later we will show how creation and termination of an object fit into an overall system.

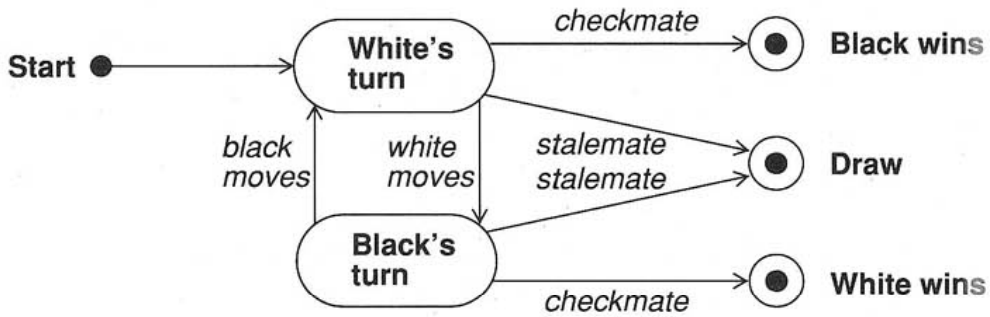

Figure 5.6 One-shot state diagram for chess game

The *dynamic model* is a collection of state diagrams that interact with each other via shared events. An object model represents the static structure of a system, while a dynamic model represents the control structure of a system. A state diagram, like an object class, is a pattern; it describes an entire, possibly infinite, range of sequences. A scenario is to a dynamic model as an instance diagram is to an object model.

5.1.5 Conditions

A *condition* is a Boolean function of object values, such as "the temperature is below freezing." A condition is valid over an interval of time. For example, "the temperature was below freezing from November 15, 1921 until March 3, 1922." It is important to distinguish conditions from events, which have no time duration. A state can be defined in terms of a condition; conversely, being in a state is a condition.

Conditions can be used as *guards* on transitions. A guarded transition fires when its event occurs, but only if the guard condition is true. For example, "when you go out in the morning (*event*), if the temperature is below freezing (*condition*), then put on your gloves (*next state*)." A guard condition on a transition is shown as a Boolean expression in brackets following the event name.

Figure 5.7 shows a state diagram with guarded transitions for traffic lights at an intersection. One pair of electric eyes checks the north-south left turn lanes; another pair checks

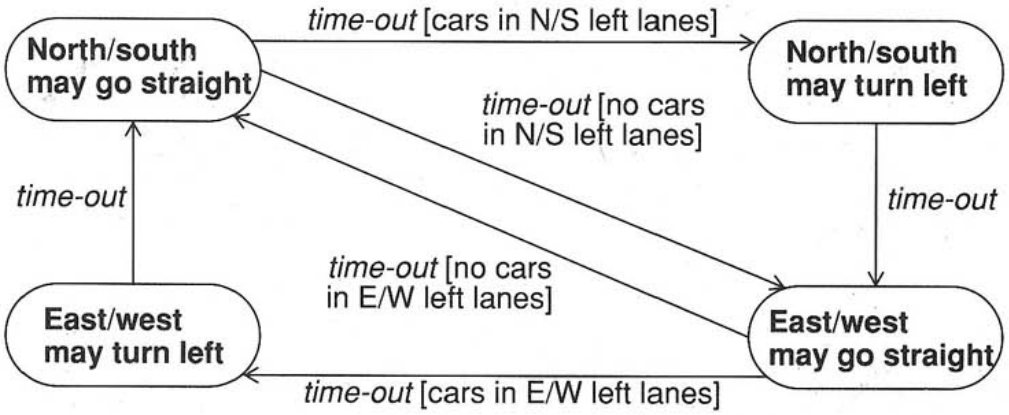

Figure 5.7 State diagram with guarded transitions

the east-west turn lanes. If no car is in the north-south and/or east-west turn lanes then the traffic light control logic is smart enough to skip the left turn portion of the cycle.

5.2 OPERATIONS

The state diagrams presented so far describe the patterns of events and states for a single object class. In this section we show how events trigger operations.

5.2.1 Controlling Operations

State diagrams would be of little use if they just described patterns of events. A behavioral description of an object must specify what the object does in response to events. Operations attached to states or transitions are performed in response to the corresponding states or events.

An *activity* is an operation that takes time to complete. An activity is associated with a state. Activities include continuous operations, such as displaying a picture on a television screen, as well as sequential operations that terminate by themselves after an interval of time, such as closing a valve or performing a computation. A state may control a continuous activity, such as ringing a telephone bell, that persists until an event terminates it by causing a transition from the state. The notation "*do: A*" within a state box indicates that activity *A* starts on entry to the state and stops on exit. A state may also control a sequential activity, such as a robot moving a part, that progresses until it completes or until it is interrupted by an event that terminates it prematurely. The same notation "*do: A*" indicates that sequential activity *A* begins on entry to the state and stops when complete. If an event causes a transition from the state before the activity is complete, then the activity is terminated prematurely. For example, a robot might encounter resistance, causing it to cease moving. The two uses are not really different; a continuous activity may be viewed as a sequential activity that lasts indefinitely.

An *action* is an instantaneous operation. An action is associated with an event. An action represents an operation whose duration is insignificant compared to the resolution of the

state diagram. For example, *disconnect phone line* might be an action in response to an *on-hook* event for the phone line in Figure 5.5. A real-world operation is not really instantaneous, of course, but modeling it as an action indicates that we do not care about its internal structure for control purposes. If we do care, then an operation should be modeled as an activity, with a starting event, ending event, and possibly some intermediate events.

Actions can also represent internal control operations, such as setting attributes or generating other events. Such actions have no real-world counterparts but instead are mechanisms for structuring control within an implementation. For example, an internal counter might be incremented every time a particular event occurs. In a computer, of course, even simple operations take some time, but they can be considered instantaneous with respect to the granularity of real events under consideration.

The notation for an action on a transition is a slash ('/') and the name (or description) of the action, following the name of the event that causes it. Figure 5.8 shows the state diagram for a pop-up menu on a workstation.When the right button is depressed, the menu is displayed; when the right button is released, the menu is erased. While the menu is visible, the highlighted menu item is updated whenever the cursor moves.

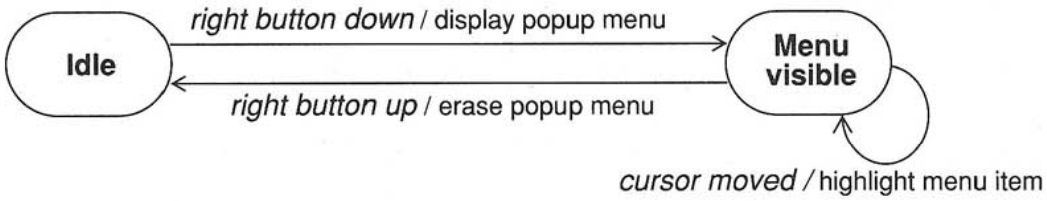

Figure 5.8 Actions for pop-up menu

5.2.2 Summary of Notation for State Diagrams with Operations

Figure 5.9 summarizes the notation presented in Sections 5.1 and 5.2 for unstructured state diagrams. Section 5.3 discusses extensions for structured state diagrams.

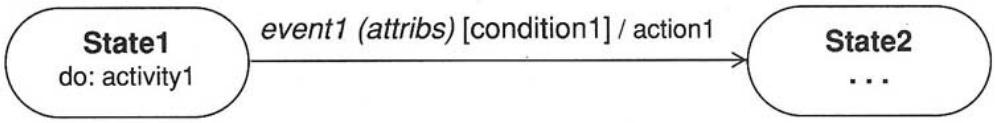

Figure 5.9 Summary of notation for unstructured state diagrams

As shown in Figure 5.9, a state name is written in boldface within a rounded box. An event name is written on a transition arrow and may optionally be followed by one or more attributes within parentheses. A condition may be listed within square brackets after an event name. An activity is indicated within a state box by the keyword "*do:*" followed by the name or description of the activity. An action is indicated on a transition following the event name by a "/" character followed by the event name. All these constructs are optional in state diagrams.

action

Figure 5.10 shows the state diagram for the phone line, previously shown in Figure 5.5, but now with actions and activities.

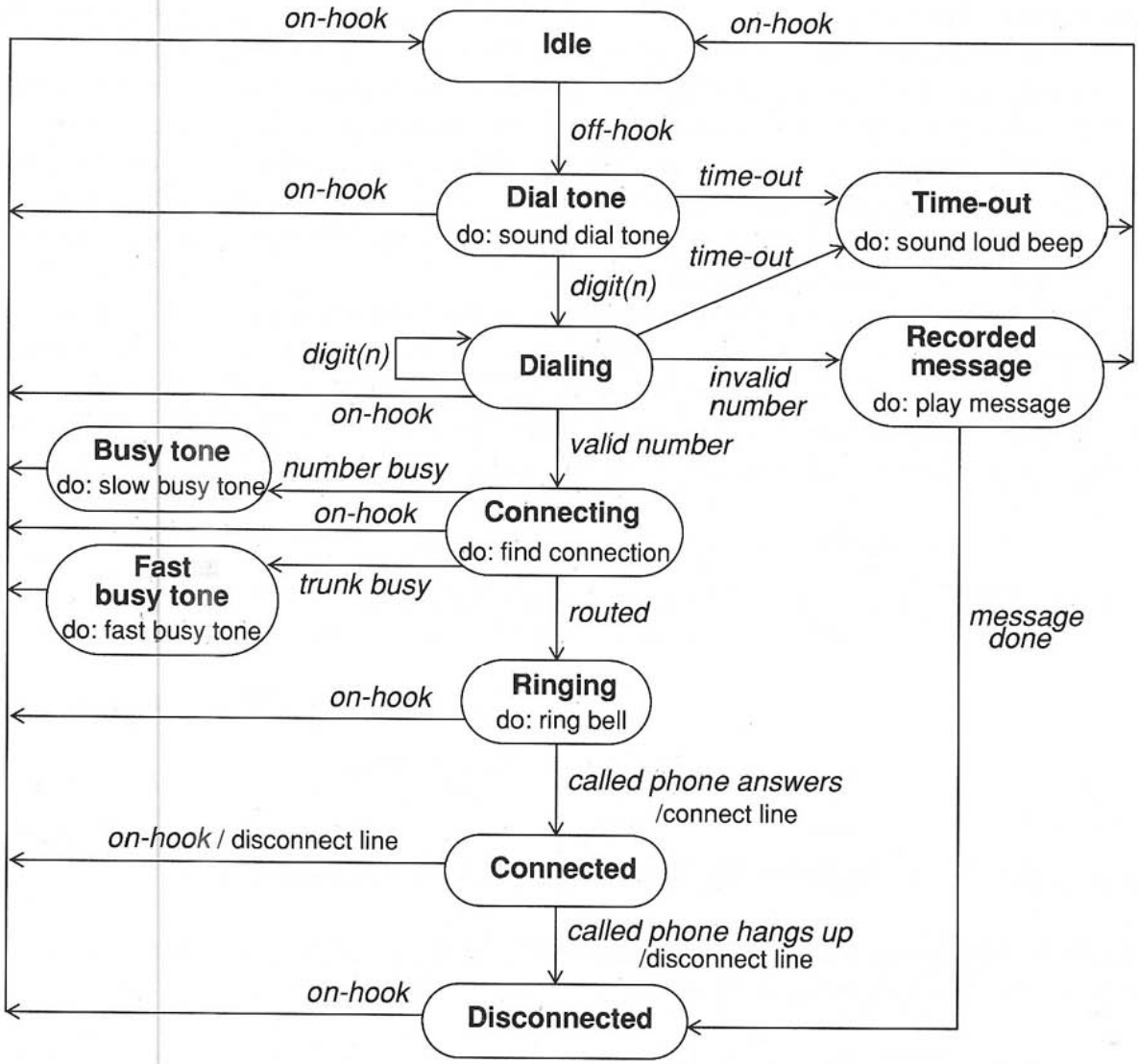

Figure 5.10 State diagram for phone line

5.3 NESTED STATE DIAGRAMS

State diagrams can be structured to permit concise descriptions of complex systems. The ways of structuring state machines are similar to the ways of structuring objects: generalization and aggregation. Generalization is equivalent to expanding nested activities. It allows an activity to be described at a high level, then expanded at a lower level by adding details, similar to a nested procedure call. In addition, generalization allows states and events to be arranged into generalization hierarchies with inheritance of common structure and behavior,

similar to inheritance of attributes and operations in classes. Aggregation allows a state to be broken into orthogonal components, with limited interaction among them, similar to an object aggregation hierarchy. Aggregation is equivalent to concurrency of states. Concurrent states generally correspond to object aggregations, possibly an entire system, that have interacting parts.

5.3.1 Problems with Flat State Diagrams

State diagrams have often been criticized because they allegedly lack expressive power and are impractical for large problems. These problems are true of flat, unstructured state diagrams. Consider an object with n independent Boolean attributes that affect control. Representing such an object with a single flat state diagram would require 2^n states. By partitioning the state into n independent state machines, however, only $2n$ states are required. Or consider the state diagram shown in Figure 5.11, in which n^2 transitions are needed to connect every state to every other state. If this model can be reformulated using structure, the number of transitions could be reduced as low as n. All complex systems contain a large amount of redundancy that can be used to simplify state diagrams, provided appropriate structuring mechanisms are available.

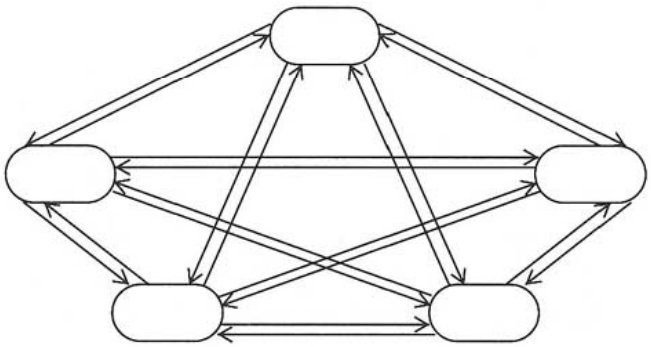

Figure 5.11 Combinatorial explosion of transitions in flat state diagram

5.3.2 Nesting State Diagrams

An activity in a state can be expanded as a lower-level state diagram, each state representing one step of the activity. Nested activities are one-shot state diagrams with input and output transitions, similar to subroutines. The set of nested state diagrams forms a lattice. (It is a tree if we expand out different copies of the same nested diagram.).

Figure 5.12 shows a top-level model for a vending machine. This diagram contains an activity *dispense item* and an event *select(item)* that are expanded in more detail in nested state diagrams. The diagram also shows a bit of alternate notation. The event *coins in(amount)* is written within the *Collecting money* state. This indicates a transition that remains within a single state. Also, the transition from the unnamed state containing "do: dispense item" to state *Idle* has no event label. The lack of event label indicates that the transition fires automatically when the activity in the state is complete.

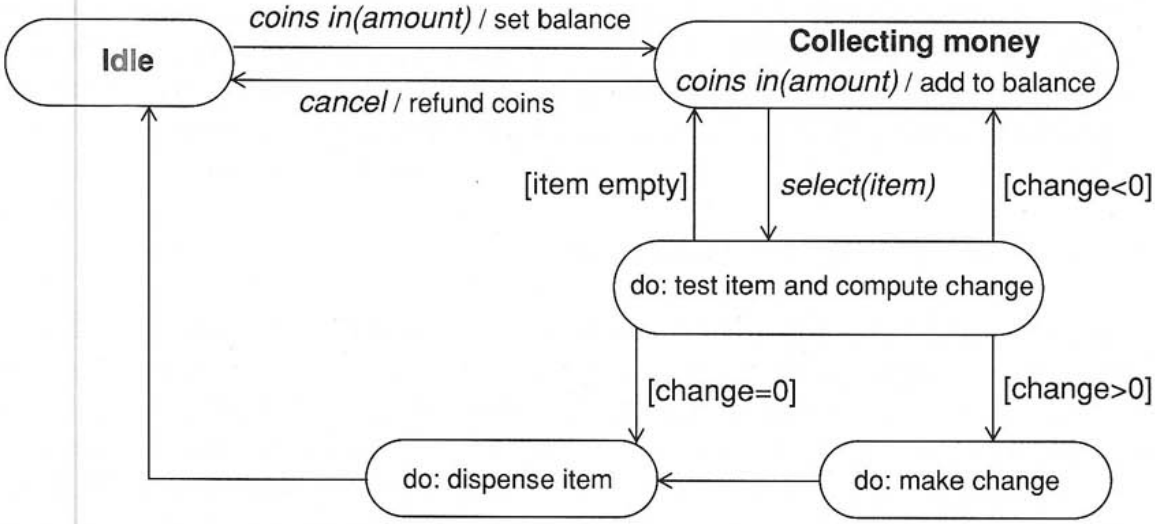

Figure 5.12 Vending machine model

Figure 5.13 shows a subdiagram for the *dispense item* activity of Figure 5.12. This activity corresponds to a sequence of lower-level states and events that are invisible in the original high-level state diagram.

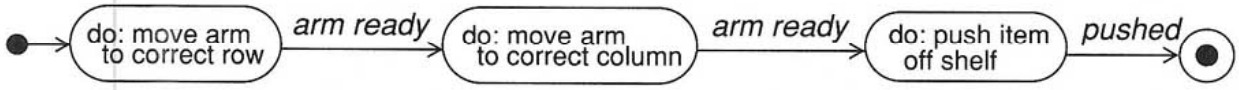

Figure 5.13 *Dispense item* activity of vending machine

Events can also be expanded into subordinate state diagrams. Figure 5.14 shows the *select item* event from Figure 5.12, which actually involves several low-level events. The customer keys in an item number and can start over by hitting *clear*; the selection is confirmed by hitting *enter*. The label on the bull's-eye indicates the event generated on the higher-level state diagram.

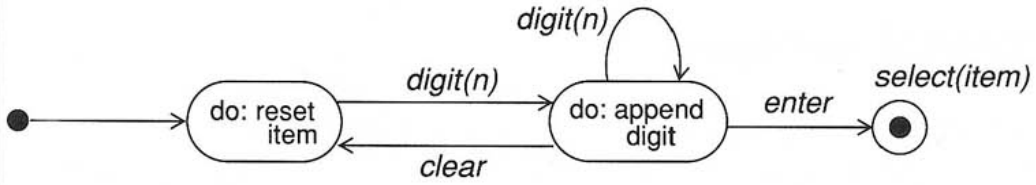

Figure 5.14 *Select item* transition of vending machine

5.3.3 State Generalization

A nested state diagram is actually a form of generalization on states. Generalization is the "or-relationship." An object in a state in the high-level diagram must be in exactly one state in the nested diagram. It must be in the first state, *or* the second state, *or* in one of the other

states. The states in the nested diagram are all refinements of the state in the high-level diagram. In the previous section, the states in the nested diagram are unaffected by transitions in the high-level diagram, but in general the states in a nested state diagram may interact with other states.

States may have substates that inherit the transitions of their superstates, just as classes have subclasses that inherit the attributes and operations of their superclasses. Any transition or action that applies to a state applies to all its substates, unless overridden by an equivalent transition on the substate. For example, the phone line model in Figure 5.5 could be simplified by replacing the transitions from each state to *Idle* on event *on-hook* with a single transition from a superstate *Active* to *Idle*. All the original states except *Idle* are substates of *Active*. The occurrence of event *on-hook* in any active substate causes a transition to state *Idle*.

Figure 5.15 shows a state diagram for an automatic transmission. The transmission can be in reverse, neutral, or forward; if it is in forward, it can be in first, second, or third gear. States *First, Second,* and *Third* are substates of state *Forward*. The generalization notation for states is different from that used for classes, to avoid a large number of lines that could be confused with transitions. A superstate is drawn as a large rounded box enclosing all of its substates. Substates in turn can enclose further substates. Because the rounded boxes representing the various states are nested, Harel calls them *contours*.

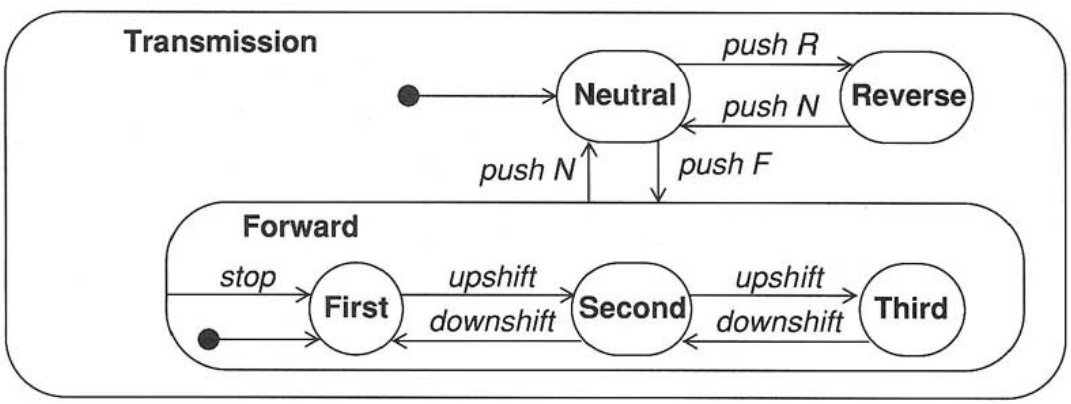

Figure 5.15 State diagram of car transmission with generalization

The transitions of a superstate are inherited by each of its substates. Selecting "N" in any forward gear causes a transition to neutral. The transition from *Forward* to *Neutral* implies three inherited transitions, one from each forward gear to neutral. Selecting "F" in neutral causes a transition to forward. Within state *Forward,* substate *First* is the default initial state, shown by the unlabeled transition from the solid circle within the *Forward* contour. *Forward* is just an abstract state; control must be in a real state, such as *First*.

The transition on event *stop* from the *Forward* contour to state *First* represents a transition inherited by all three substates. In any forward gear, stopping the car causes a transition to *First*.

It is possible to represent more complicated situations, such as an explicit transition from a substate to a state outside the contour, or an explicit transition into the contour. In such cases, all the states must appear on one diagram using the contour notation. In simpler cases

where there is no interaction except for initiation and termination, the nested states can simply be drawn as a separate diagram and referenced by name in a "do" statement, as in the vending machine example of Figure 5.12.

5.3.4 Event Generalization

Events can be organized into a generalization hierarchy with inheritance of event attributes. Figure 5.16 shows part of a tree of input events for a workstation. Events *mouse button down* and *keyboard character* are two kinds of user input. Both events inherit attribute *time* from event *event* (the root of the hierarchy) and attribute *device* from event *user input*. *Mouse button down* and *mouse button up* inherit *location* from *mouse button*. Keyboard characters can be divided into control characters and graphic characters. Ultimately every actual event can be viewed as a leaf on a generalization tree of events. Inherited event attributes are shown in the second part of each box. An input event triggers transitions on any ancestor event type. For example, typing an 'a' would trigger a transition on event *alphanumeric* as well as event *keyboard character*.

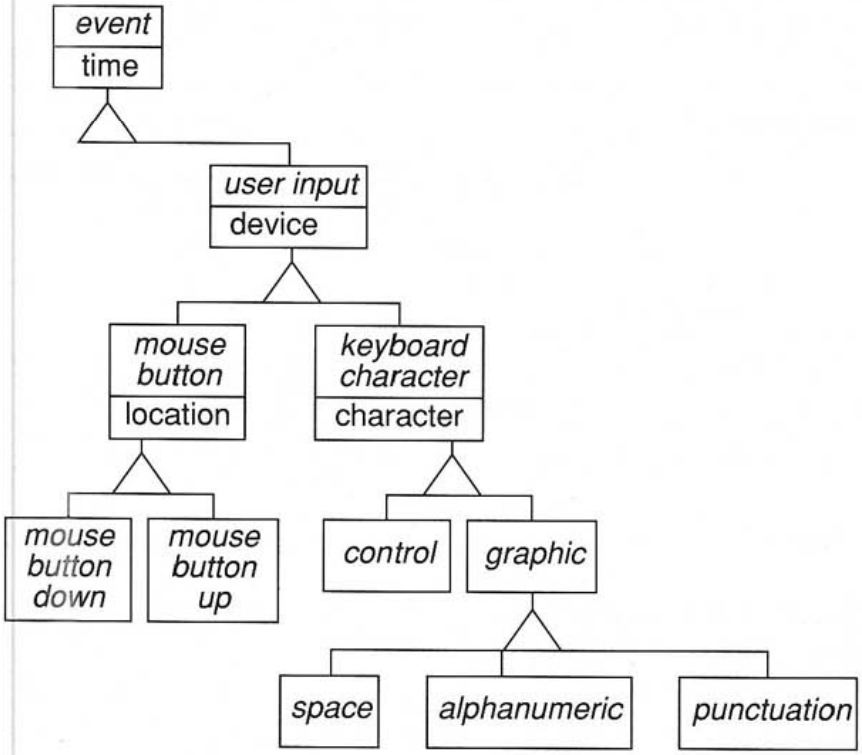

Figure 5.16 Partial event hierarchy for keyboard events

Providing an event hierarchy permits different levels of abstraction to be used at different places in a model. For example, in some states all input characters might be handled the same and would lead to the same next state; in other states control characters would be treated differently from printing characters; still others might have different actions on individual characters.

5.4 CONCURRENCY

5.4.1 Aggregation Concurrency

A dynamic model describes a set of concurrent objects, each with its own state and state diagram. The objects in a system are inherently concurrent and can change state independently. The state of the entire system cannot be represented by a single state in a single object; it is the product of the states of all the objects in it. In many systems, the number of objects can change dynamically as well.

A state diagram for an assembly is a collection of state diagrams, one for each component. Aggregation implies concurrency. The aggregate state corresponds to the combined states of all the component diagrams. Aggregation is the "and-relationship." The aggregate state is one state from the first diagram, *and* a state from the second diagram, *and* a state from each other diagram. In the more interesting cases, the component states interact. Guarded transitions for one object can depend on another object being in a given state. This allows interaction between the state diagrams, while preserving modularity.

Figure 5.17 shows the state of a *Car* as an aggregation of component states: the *Ignition*, *Transmission*, *Accelerator*, and *Brake* (plus other unmentioned objects). Each component state also has substates. The state of the car includes one substate from each component. Each component undergoes transitions in parallel with all the others. The state diagrams of the components are almost, but not quite, independent: The car will not start unless transmission is in neutral. This is shown by the guard expression *Transmission in Neutral* on the transition from *Ignition-Off* to *Ignition-Starting*.

5.4.2 Concurrency within an Object

Concurrency within the state of a single object arises when the object can be partitioned into subsets of attributes or links, each of which has its own subdiagram. The state of the object comprises one state from each subdiagram. The subdiagrams need not be independent; the same event can cause transitions in more than one subdiagram. Concurrency within a single composite state of an object is shown by partitioning the composite state into subdiagrams with dotted lines. The name of the overall composite state can be written in a separate region of the box, separated by a solid line from the concurrent subdiagrams. Figure 5.18 shows the state diagram for the play of a bridge rubber. When a side wins a game, it becomes "vulnerable"; the first side to win two games wins the rubber. During the play of the rubber, the state of the rubber consists of one state from each subdiagram. When the *Playing rubber* composite state is entered, both subdiagrams are initially in their respective default states *Not vulnerable*. Each subdiagram can independently advance to state *Vulnerable* when its side wins a game. When one side wins a second game, a transition occurs to the corresponding *Wins rubber* state. This transition terminates both concurrent subdiagrams because they are part of the same composite state *Playing rubber* and are only active when the top-level state diagram is in that state.

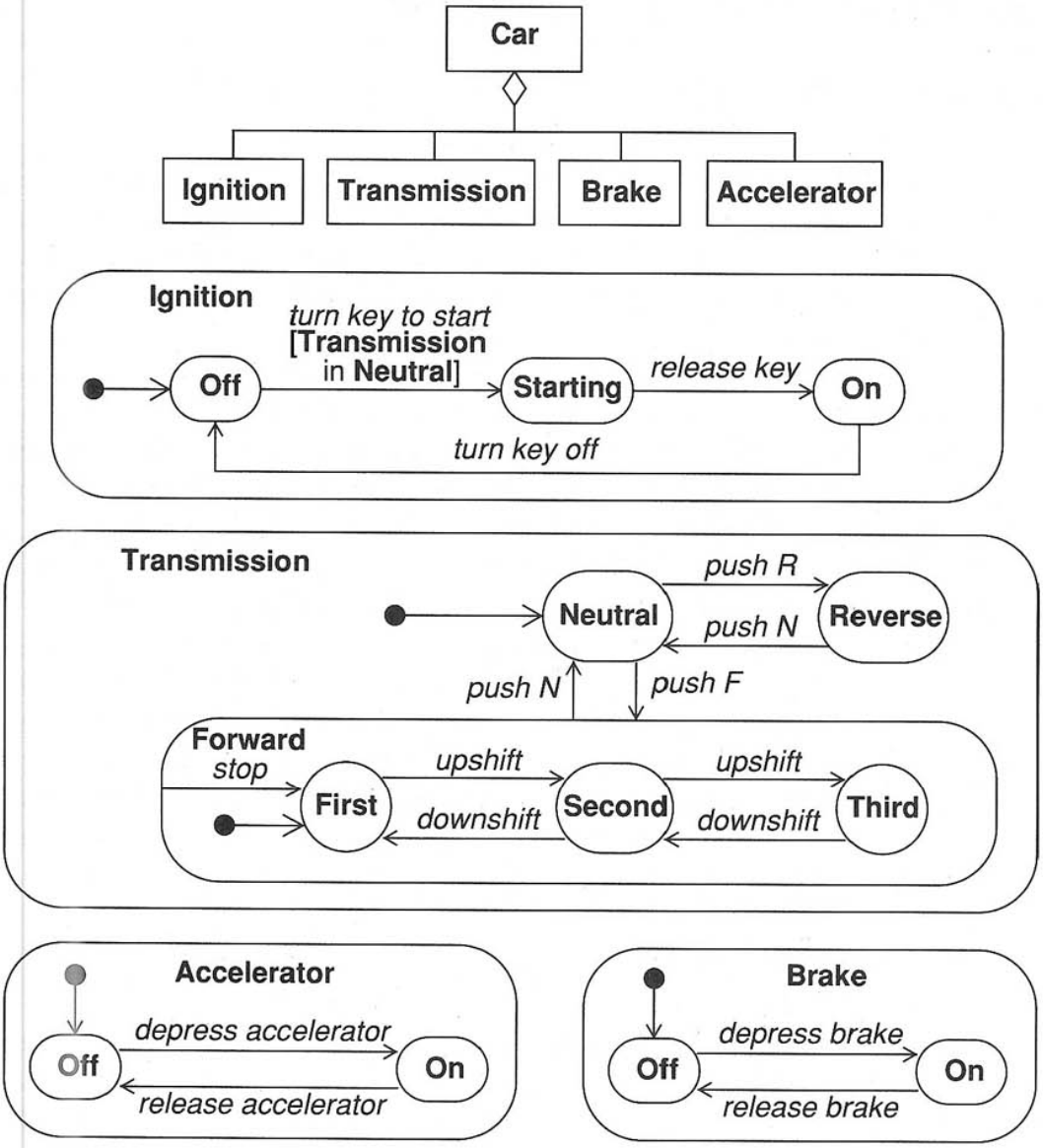

Figure 5.17 An aggregation and its concurrent state diagrams

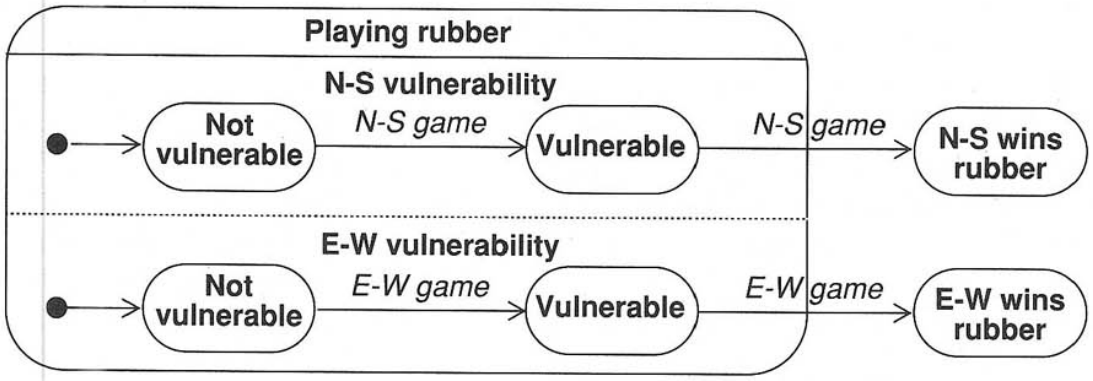

Figure 5.18 Bridge game with concurrent states

5.5 ADVANCED DYNAMIC MODELING CONCEPTS

In this section we present advanced dynamic modeling concepts as well as some refinements on the notation.

5.5.1 Entry and Exit Actions

As an alternative to showing actions on transitions, actions can be associated with entering or exiting a state. There is no difference in expressive power between the two notations, but frequently all transitions into a state perform the same action, in which case attaching the action to the state is more concise.

For example, Figure 5.19 shows the control of a garage door opener. The user generates *depress* events with a push-button to open and close the door. Each event reverses the direction of the door, but for safety the door must open fully before it can be closed. The control generates *motor up* and *motor down* actions for the motor. The motor generates *door open* and *door closed* events when the motion has been completed. Both transitions entering state *Opening* cause the door to open.

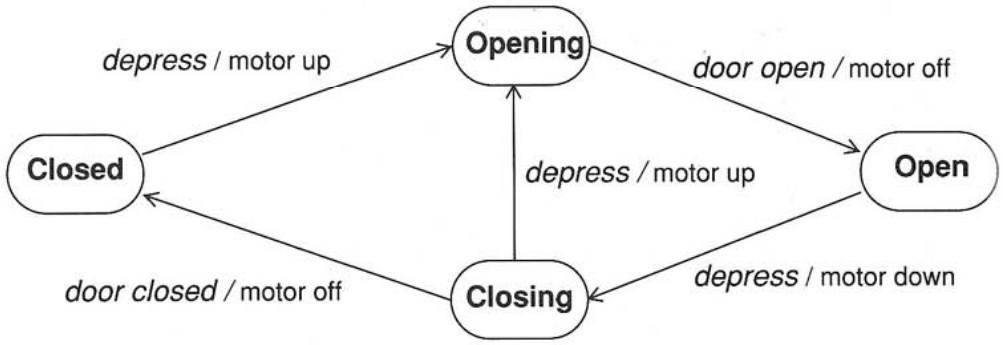

Figure 5.19 Actions on transitions

Figure 5.20 shows the same model using actions on entry to states. An entry action is shown inside the state box following the keyword *entry* and a "/" character. Whenever the state is entered, by any incoming transition, the entry action is performed. An entry action is equivalent to attaching the action to every incoming transition. If an incoming transition already has an action, its action is performed first.

Exit actions are less common than entry actions, but they are occasionally useful. An exit action is shown inside the state box following the keyword *exit* and a "/" character. Whenever the state is exited, by any outgoing transition, the exit action is performed first.

If multiple operations are specified on a state, they are performed in the following order: actions on the incoming transition, entry actions, do activities, exit actions, actions on the outgoing transition. Do activities can be interrupted by events that cause transitions out of the state, but entry actions and exit actions are completed regardless, since they are considered to be instantaneous actions. If a do activity is interrupted, the exit action is nevertheless performed.

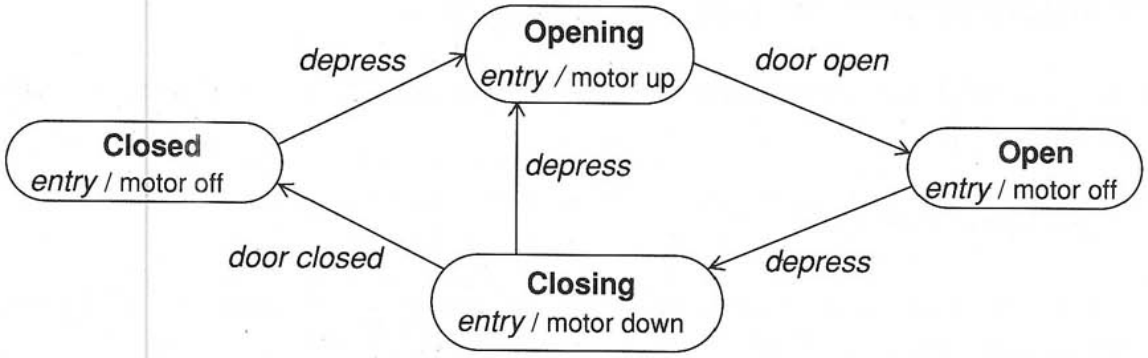

Figure 5.20 Actions on entry to states

Entry and exit actions are particularly useful in nested state diagrams because they permit a state (possibly an entire subdiagram) to be expressed in terms of matched entry-exit actions without regard for what happens before or after the state is active. It is possible to use actions attached to transitions as well as entry and exit actions in a diagram.

Transitioning into or out of a substate in a nested diagram can cause execution of several entry or exit actions, if the transition reaches across several levels of generalization. The entry actions are executed from the outside in and the exit actions from the inside out. This permits behavior similar to nested subroutine calls.

5.5.2 Internal Actions

An event can cause an action to be performed without causing a state change. The event name is written inside the state box, followed by a "/" and the name of the action. (Keywords *entry, exit,* and *do* are reserved words within the state box.) When such an event occurs, its action is executed but not the entry or exit actions for the state. There is therefore a difference between an internal action and a self-transition; the self-transition causes the exit and entry actions for the state to be executed. Figure 5.12 shows an internal action within the *Collecting money* state.

Figure 5.21 summarizes the additional notation for entry, exit, and internal actions.

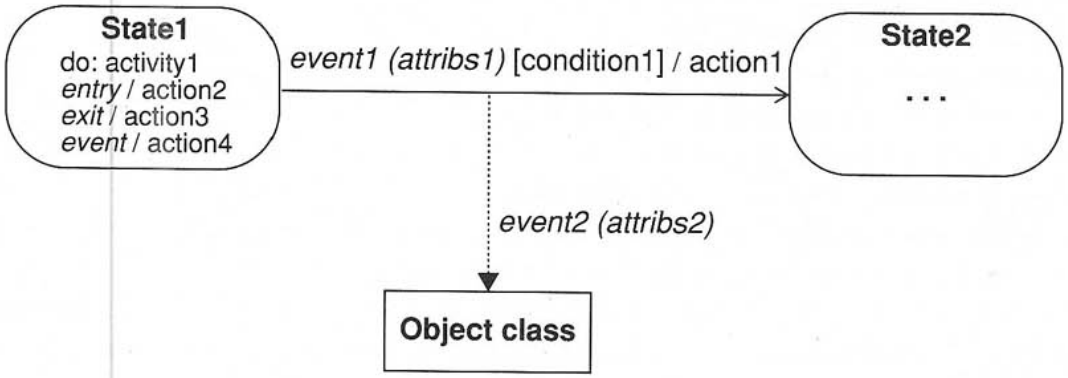

Figure 5.21 Summary of extended notation for state diagrams

5.5.3 Automatic Transition

Frequently the only purpose of a state is to perform a sequential activity. When the activity is completed, a transition to another state fires. An arrow without an event name indicates an automatic transition that fires when the activity associated with the source state is completed. If there is no activity, the unlabeled transition fires as soon as the state is entered (but the entry and exit actions are always performed). Such unlabeled transitions are sometimes called *lambda transitions,* after the Greek letter used to indicate them in some textbooks. Figure 5.12 shows four unlabeled transitions from the state containing activity "test item and compute change." Each transition has a guard condition. When the activity is complete, the transition with a valid guard condition fires.

If a state has one or more automatic transitions, but none of the guard conditions are satisfied, then the state remains active until one of the conditions is satisfied or until an event causes another transition to fire. The change in value of a condition is an implicit event (referred to in digital hardware as "edge triggering"). For example, "the temperature is below freezing" is a condition. "The temperature goes belong freezing" is the edge-triggered event associated with the condition.

5.5.4 Sending Events

An object can perform the action of sending an event to another object. A system of objects interacts by exchanging events.

The action *"send E(attributes)"* sends event E with the given attributes to the object or objects that receive it. For example, the phone line sends a *connect(phone-number)* event to the switcher when a complete phone number has been dialed. An event can be directed at a set of objects or a single object. Any and all objects with transitions on the event can accept it concurrently. The word "send" can be omitted if it is clear that E is the name of an event. In our diagrams, event names are shown in italics and action names in normal text, so there is no confusion.

Figure 5.21 shows another notation for sending an event from one object to another. The dotted line from a transition to an object indicates that an event is sent to the object when the transition fires. The arrow could be connected directly to a transition within the state diagram of the target object to indicate that the target transition depends on the event.

If a state can accept events from more than one object, the order in which concurrent events are received may affect the final state; this is called a *race condition.* For example, in Figure 5.20 the door may or may not remain open if the button is pressed at about the time the door becomes fully open. A race condition is not necessarily a design error, but concurrent systems frequently contain unwanted race conditions which must be avoided by careful design. A requirement of two events being received simultaneously is never a meaningful condition in the real world, as slight variations in transmission speed are inherent in any distributed system.

When an object interacts with an external object, such as a person or device, sending an event is often indistinguishable from an action. For example, in the event trace of Figure 5.3,

actions *dial tone begins* and *ringing tone* are actually events between the phone line and the caller.

5.5.5 Synchronization of Concurrent Activities

Sometimes one object must perform two (or more) activities concurrently. The internal steps of the activities are not synchronized, but both activities must be completed before the object can progress to its next state. For example, consider a cash dispensing machine that dispenses cash and returns the user's card at the end of a transaction. The machine must not reset itself until the user takes both the cash and the card, but the user may take them in either order or even simultaneously. The order in which they are taken is irrelevant, only the fact that both of them have been taken. This is an example of *splitting control* into concurrent activities and later *merging control*.

Figure 5.22 shows a concurrent state diagram for the emitting activity. Concurrent activities within a single composite activity are shown by partitioning a state into regions with dotted lines, as explained previously. Each region is a subdiagram that represents a concurrent activity within the composite activity. The composite activity assumes exactly one state from each subdiagram.

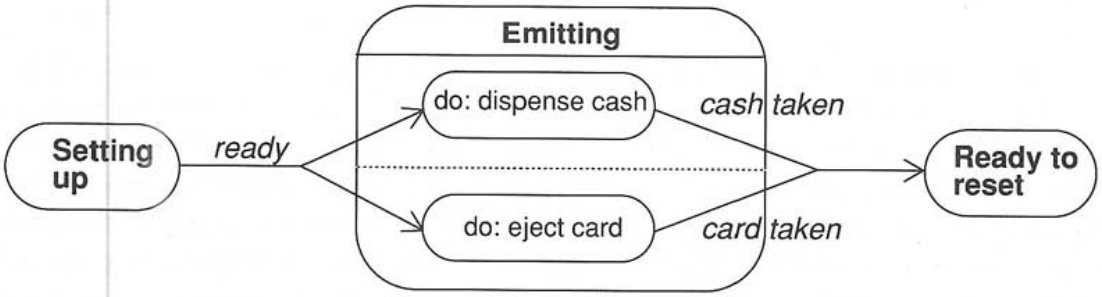

Figure 5.22 Synchronization of control

Splitting of control into concurrent parts is shown by an arrow that forks. The forked arrow selects one state from each concurrent subdiagram. In the example, the transition on event *ready* splits into two concurrent parts, one to each concurrent subdiagram. When this transition fires, two concurrent substates become active and execute independently. Each concurrent substate could be a whole state diagram.

Any transition into a state with concurrent subdiagrams activates each of the subdiagrams. If any subdiagrams are omitted from the transition, they start in their default initial states. In this example, a forked arrow is not actually necessary. A transition could be drawn to the *Emitting* state, with a default initial state indicated in each subdiagram.

Merging of concurrent control is shown by an arrow with a forked tail. The target state becomes active when both events occur in any order. The events need not be simultaneous. Each subdiagram terminates as soon as its part of the transition fires, but all parts of the transition must fire before the entire transition fires and the composite state is terminated. If there are any subdiagrams in the composite state that are not part of the merge, then they are

automatically terminated when the merge transition fires. The exit actions (if any) of all subdiagrams are performed when the merge transition fires. In the example, the transitions *cash taken* and *card taken* are part of a single merge transition. When both parts of the merge transitions fire, state *Ready to reset* becomes active. Drawing a separate transition from each substate to the target state would have a different meaning; either transition would terminate the other subdiagram without waiting for the other transition.

In this example, the number of concurrently-active states varies during execution from one to two and back to one again.

5.6 A SAMPLE DYNAMIC MODEL

We present a sample dynamic model of a real device to show how the various modeling constructs fit together. This is a model of a Sears "Weekender" Programmable Thermostat. This model was constructed by reading the instruction manual and by experimenting with the actual device. This device controls a furnace and air conditioner according to time-dependent attributes which the owner enters using a pad of buttons.

While running, the thermostat operates the furnace or air conditioner to keep the current temperature equal to the target temperature. The target temperature is taken from a table of program values supplied by the user. The table specifies target temperature for 8 different time periods, 4 on weekdays and 4 on weekends, with start times specified by the user. The target temperature is reset from the table at the beginning of each program period. The user can override the target temperature for the remainder of the current period or indefinitely. The user programs the thermostat using a pad of 10 push buttons and 3 switches. The user sees parameters on an alphanumeric display. A switch illuminates a night light. The thermostat has a temperature sensor that reads the air temperature. The thermostat operates power relays for a furnace and an air conditioner, and an indicator lights up when the furnace or air conditioner is operating.

Each push button generates an event every time it is pushed. We assign one input event per button:

TEMP UP	raises target temperature or program temperature
TEMP DOWN	lowers target temperature or program temperature
TIME FWD	advances clock time or program time
TIME BACK	retards clock time or program time
SET CLOCK	sets current time of day
SET DAY	sets current day of the week
RUN PRGM	leaves setup or program mode and runs the program
VIEW PRGM	enters program mode to examine and modify 8 program time and program temperature settings
HOLD TEMP	holds current target temperature in spite of the program
F-C BUTTON	alternates temperature display between Fahrenheit and Celsius

Each switch supplies a parameter value chosen from two or three possibilities. We model each switch as an independent concurrent subdiagram with one state per switch setting. Although we assign event names to a change in state, it is the state of each switch that is of interest. The switches and their settings are:

Light switch	Lights the alphanumeric display. Values: light off, light on.
Season switch	Specifies which device the thermostat controls. Values: heat (furnace), cool (air conditioner), off (none).
Fan switch	Specifies when the ventilation fan operates. Values: fan on (fan runs continuously), fan auto (fan runs only when furnace or air conditioner is operating).

The thermostat controls the furnace, air conditioner, and fan power relays. We model this control by activities "run furnace," "run air conditioner," and "run fan."

The thermostat has a temperature sensor that it reads continuously, which we model by an external parameter *temp*. The thermostat also has an internal clock that it reads and displays continuously. We model the clock as another external parameter *time*, since we are not interested in building a state model of the clock. In building a dynamic model, it is important to only include states that affect the flow of control and to model other information as parameters or variables. We introduce an internal state variable *target temp*, which represents the current temperature that the thermostat is trying to maintain. This state variable is read by some actions and set by other actions; it permits communication among parts of the dynamic model.

Figure 5.23 shows the top-level state diagram of the programmable thermostat. It contains 7 concurrent subdiagrams. The user interface is too large to show and is expanded separately. The diagram includes trivial subdiagrams for the season switch and the fan switch. The other 4 subdiagrams show the output of the thermostat: the furnace, air conditioner, and fan relays, and the run indicator light. Each of these subdiagrams contains an *Off* and an *On* substate. The state of each subdiagram is totally determined by conditions on input parameters and the state of other subdiagrams, such as the season switch or the fan switch. The state of the 4 subdiagrams on the right is totally derived and contains no additional information.

Figure 5.24 shows the subdiagram for the user interface. The diagram contains 3 concurrent subdiagrams, one for the interactive display, one for the temperature mode, and one for the night light. The night light is controlled by a physical switch, so the default initial state is irrelevant; its value can be determined directly. The temperature display mode is controlled by a single push button that toggles the temperature units between Fahrenheit and Celsius. The default initial state is necessary; when the device is powered on, the initial temperature mode is Fahrenheit.

The subdiagram for the interactive display is more interesting. The device is either operating or being set up. Substate *Operate* contains two substates, *Run* and *Hold*, in addition to two concurrent substates, one which controls the target temperature display and one which controls the current time and temperature display. Every 2 seconds the display alternates between the current time and current temperature.

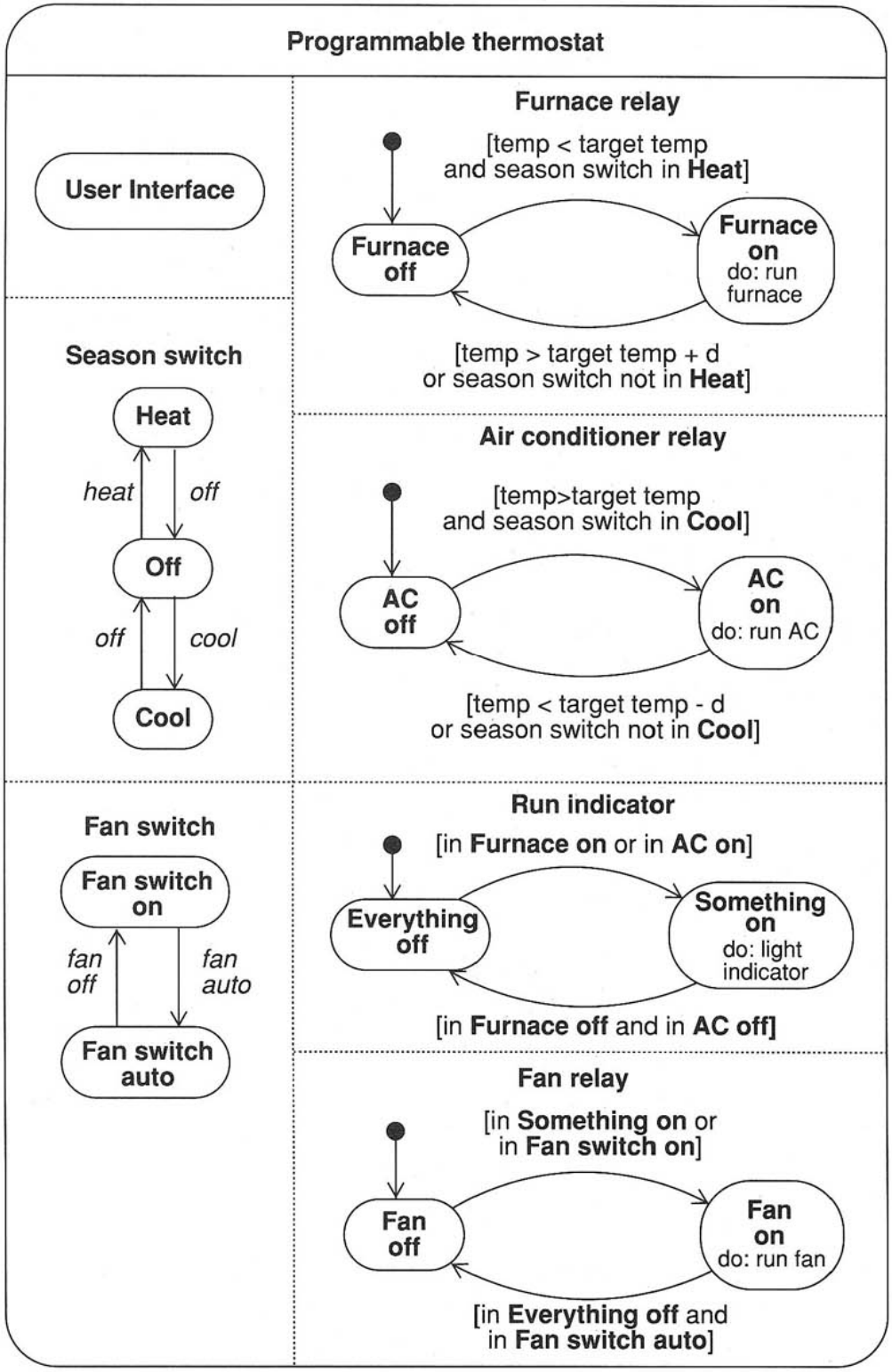

Figure 5.23 State diagram for programmable thermostat

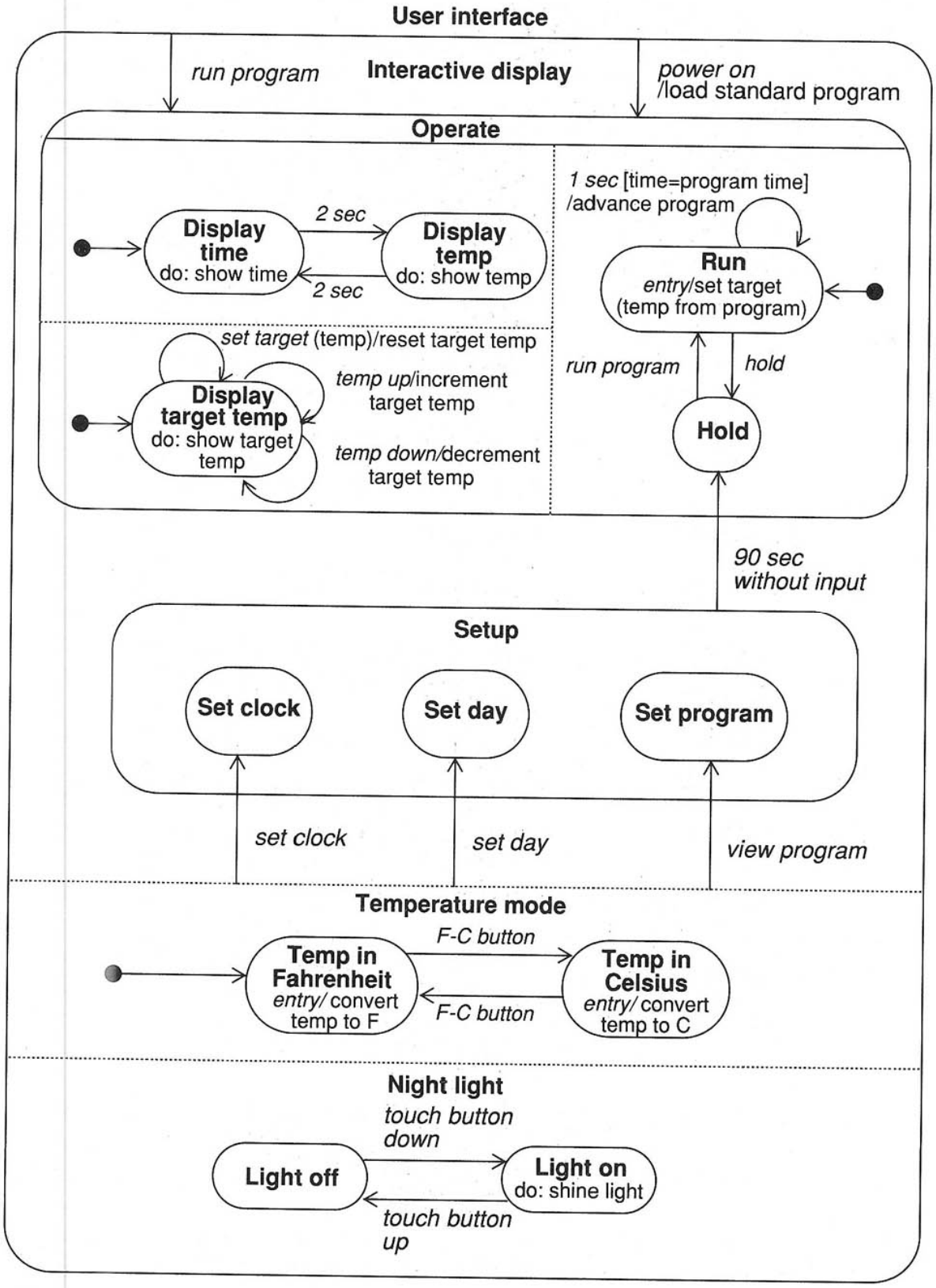

Figure 5.24 Subdiagram for thermostat user interface

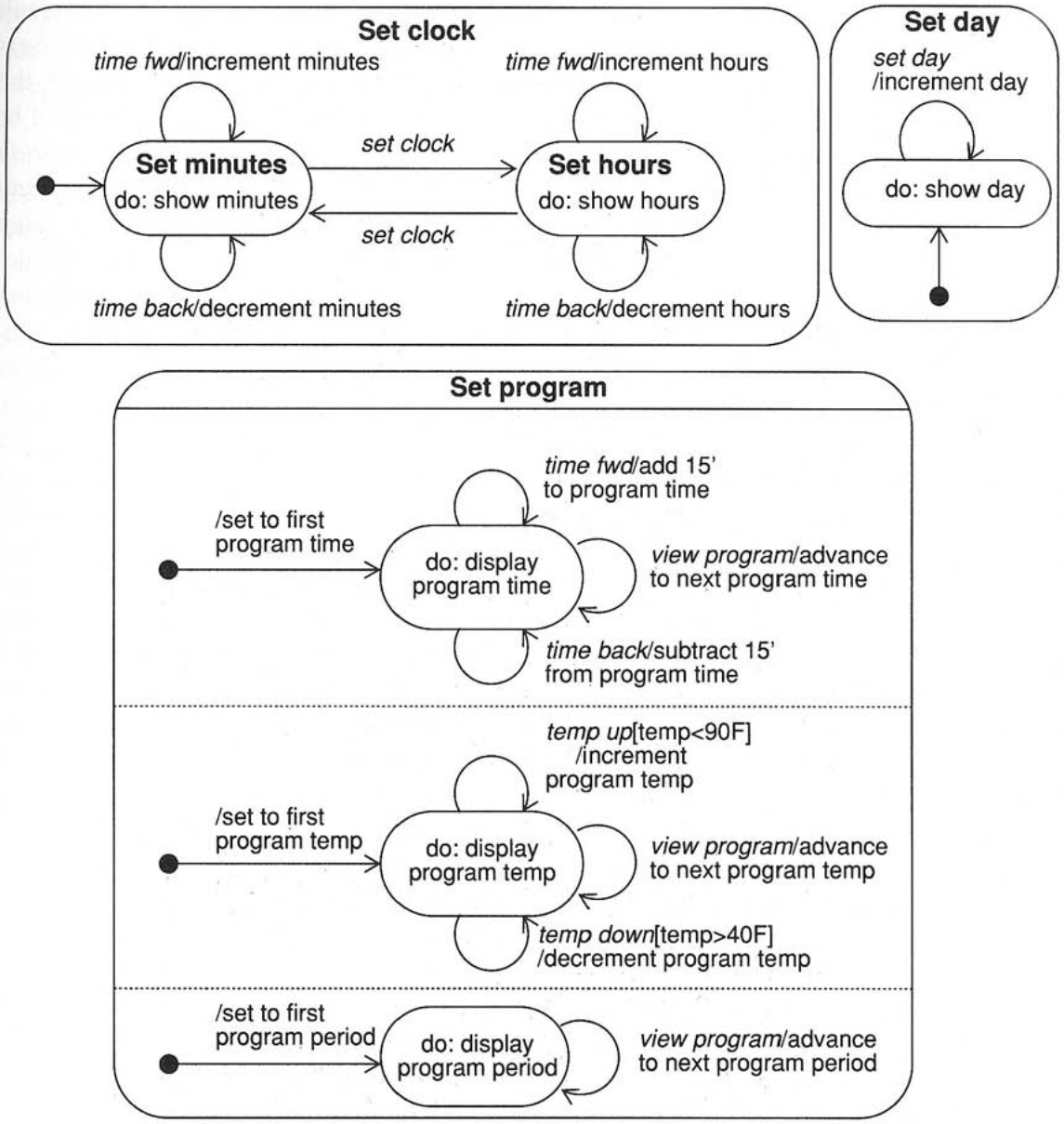

Figure 5.25 Subdiagrams for thermostat user interface setup

The target temperature is displayed continuously and is modified by the *temp up* and *temp down* buttons, as well as the *set target* event that is generated only in the *Run* state. Note that the *target temp* parameter set by this subdiagram is the same parameter that controls the output relays.

While in the *Operate* state, the device is either in the *Run* or *Hold* substates. Every second in the *Run* state, the current time is compared to the stored program times in the program table; if they are equal, then the program advances to the next program period, and the *Run* state is reentered. The run state is also entered whenever the *run program* button is pressed

in any state, as shown by the transition from the contour to the *Operate* state and the default initial transition to *Run*. Whenever the *Run* state is entered, the entry action on the state resets the target temperature from the program table. While the program is in the *Hold* state, the program temperature cannot be advanced automatically, but the temperature can still be modified directly by the *temp up* and *temp down* buttons. The default substate on *power on* is the *Run* substate. If the interface is in one of the setup states for 90 seconds without any input, the system enters the *Hold* state. This transition is shown as an arrow from the *Setup* contour directly to the *Hold* substate. Entering the *Hold* substate also forces entry to the default initial states of the other two concurrent subdiagrams of *Operate*. The *Setup* state was included in the model just to group the three setup substates for the 90-second time-out transition. Note a small anomaly of the device: The *hold* button has no effect within the *Setup* state, although the *Hold* state can be entered by waiting for 90 seconds.

The three setup subdiagrams are shown in Figure 5.25. Pressing *set clock* enters the *Set minutes* substate as initial default. Subsequent *set clock* presses toggle between the *Set hours* and the *Set minutes* substates. The *time fwd* and *time back* buttons modify the program time. Pressing *set day* enters the *Set day* substate and shows the day of the week. Subsequent presses increment the day directly. Pressing *view program* enters the *Set program* substate, which has three concurrent subdiagrams, one each controlling the display of the program time, program temperature, and program period. The *Set program* state always starts with the first program period, while subsequent *view program* events cycle through the 8 program periods. The *view program* event is shown on all three subdiagrams, each diagram advancing the setting that it controls. Note that the *time fwd* and *time back* events modify time in 15 minute increments, unlike the same events in the *set clock* state. Note also that the *temp up* and *temp down* transitions have guard conditions to keep the temperature in a fixed range.

None of the *Interactive display* substates has an explicit exit transition. Each substate is implicitly terminated by a transition into another substate from the main *Interactive display* contour.

5.7 RELATION OF OBJECT AND DYNAMIC MODELS

The dynamic model specifies allowable sequences of changes to objects from the object model. A state diagram describes all or part of the behavior of one object of a given class. States are equivalence classes of attribute and link values for the object. Events can be represented as operations on the object model.

Dynamic model structure is related to and constrained by object model structure. A substate refines the attribute and link values that the object can have. Each substate restricts the values that the object can have. But this refinement of object values is exactly generalization by restriction, as discussed in Section 4.3. A hierarchy of states of an object is equivalent to a restriction hierarchy of the object class. Object-oriented models and languages do not usually support restriction in the generalization hierarchy, so the dynamic model is the proper place to represent it. Both generalization of classes and generalization of states partition the set of possible object values. A single object can have different states over time—the object

preserves its identity—but it cannot have different classes. Inherent differences among objects are therefore properly modeled as different classes, while temporary differences are properly modeled as different states of the same class.

A composite state is the aggregation of more than one concurrent substate. There are three sources of concurrency within the object model. The first is aggregation of objects: Each component of an aggregation has its own independent state, so the assembly can be considered to have a state that is the composite of the states of all its parts. The second source is aggregation within an object: The attributes and links of an object are its parts, and groups of them taken together define concurrent substates of the composite object state. The third source is concurrent behavior of an object, such as found in Figure 5.22. The three sources of concurrency are usually interchangeable. For example, an object could contain an attribute to indicate that it was performing a certain activity.

The dynamic model of a class is inherited by its subclasses. The subclasses inherit both the states of the ancestor and the transitions. The subclasses can have their own state diagrams. But how do the state diagrams of the superclass and the subclass interact? We have noted that states are equivalent to restriction on classes. If the superclass state diagrams and the subclass state diagrams deal with disjoint sets of attributes, there is no problem. The subclass has a composite state composed of concurrent state diagrams. If, however, the state diagram of the subclass involves some of the same attributes as the state diagram of the superclass, a potential conflict exists. The state diagram of the subclass must be a refinement of the state diagram of the superclass. Any state from the parent state diagram can be generalized or split into concurrent parts, but new states or transitions cannot be introduced into the parent diagram directly because the parent diagram must be a projection of the child diagram.Although refinement of inherited state diagrams is possible, usually the state diagram of a subclass should be an independent, orthogonal, concurrent addition to the state diagram inherited from a superclass, defined on a different set of attributes (usually the ones added in the subclass).

The event hierarchy is independent of the class hierarchy, in practice if not in theory. Events can be defined across different classes of objects. Events are more fundamental than states and more parallel to classes. States are defined by the interaction of objects and events. Transitions can often be implemented as operations on objects. The operation name corresponds to the event name. Events are more expressive than operations, however, because the effect of an event depends not only on the class of an object but also on its state.

5.8 PRACTICAL TIPS

The precise content of all OMT models depends on the needs of the application. This is true for the object, dynamic, and functional models. The examples in this chapter illustrate the various modeling constructs without showing the process for constructing a model in the first place. Parts 2 and 3 of this book show how to apply these principles; Part 4 presents several real applications.

The following practical tips have been mentioned throughout the chapter but are summarized here for convenience.

- Only construct state diagrams for object classes with meaningful dynamic behavior. Not all object classes require a state model.
- Check the various state diagrams for consistency on shared events so that the full dynamic model will be correct.
- Use scenarios to help you begin the process of constructing state diagrams. (Chapter 8 describes this process in detail.)
- Only consider *relevant* attributes when defining a state. All attributes shown in an object model need not be used in state diagrams.
- Consider the needs of the application when deciding on the granularity of events and states.
- Let the application distinguish between activities and actions. Activities occur over a period of time; actions are instantaneous compared to the time scale of an application.
- When a state has multiple incoming transitions and all transitions cause the same action to occur, put actions within state boxes preceded by an *entry* event instead of listing them on transition arcs. Do likewise for *exit* events.
- Use nested states when the same transaction applies to many states.
- Most concurrency arises from object aggregation and need not be expressed explicitly in the state diagram. Use composite states to show independent facets of the behavior of a single object.
- Try to make the state diagrams of subclasses independent of the state diagrams of their superclasses. The subclass state diagrams should concentrate on attributes unique to the subclasses.
- Beware of unwanted race conditions in state diagrams. Race conditions may occur when a state can accept events from more than one object.

5.9 CHAPTER SUMMARY

The dynamic model represents control information: the sequences of events, states, and operations that occur within a system of objects. Like the object model, the dynamic model is a pattern that specifies the allowable scenarios that may occur. The notation for the dynamic model represents a compromise between simplicity and expressiveness; there are some meaningful constraints that cannot be expressed by the notation we present. As with the object model, these constraints must be expressed in natural language.

An event is a signal that something has happened. A state represents the interval between events and specifies the context in which events are interpreted. A transition between states represents the response to an event, including the next state and possible actions and

events sent to other objects. A condition is a Boolean function that controls whether a transition is allowed to occur. A state diagram is a graph of states and transitions labeled by events.

An action is an instantaneous operation in response to an event, often purely formal or internal. One kind of action is sending an event to another object. Actions can be attached to transitions or to entering or exiting a state. An activity is a sequence of actions that takes time to complete. An activity can be equated with a state or an entire state diagram. The result of an activity can be used as a decision to choose the next state.

States and events can both be expanded into nested state diagrams to show greater detail. Events and states can both be organized into inheritance hierarchies. Substates inherit the transitions of their superstates. Subevents trigger the same transitions as their superevents.

Objects are inherently concurrent. Each object is a collection that has its own state. State diagrams show concurrency as an aggregation of concurrent states, each operating independently. Concurrent objects interact by exchanging events and by testing conditions of other objects, including states. Transitions can split or merge flow of control.

Entry and exit actions permit actions to be associated with a state, to indicate all the transitions entering or exiting the state. They make self-contained state diagrams possible for use in multiple contexts. Internal actions represent transitions that do not leave the state. Automatic transitions fire when the their conditions are satisfied and any activity in the source state has terminated.

States are really restriction generalizations on a class and are complementary to ordinary extension generalizations. A subclass inherits the state diagrams of its ancestors, to be concurrent with any state diagram that it defines. It is also possible to refine an inherited state diagram by expanding states into substates or concurrent subdiagrams.

A realistic model of a programmable thermostat takes three pages and illustrates subtleties of behavior that are not apparent from the instruction manual or from everyday operation.

action	contour	generalization	scenario
activity	control	guard	state
aggregation	dynamic model	nested diagram	state diagram
concurrency	event	operation	transition
condition	event trace	race condition	

Figure 5.26 Key concepts for Chapter 5