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.

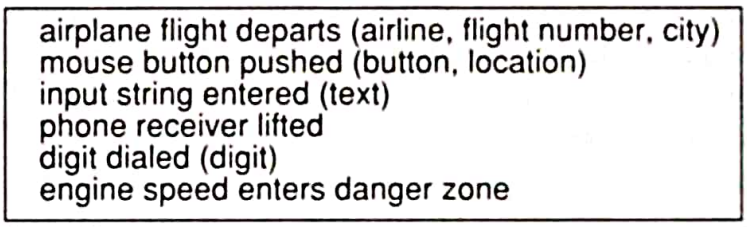

Figure 5.1 Event classes and attributes

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.

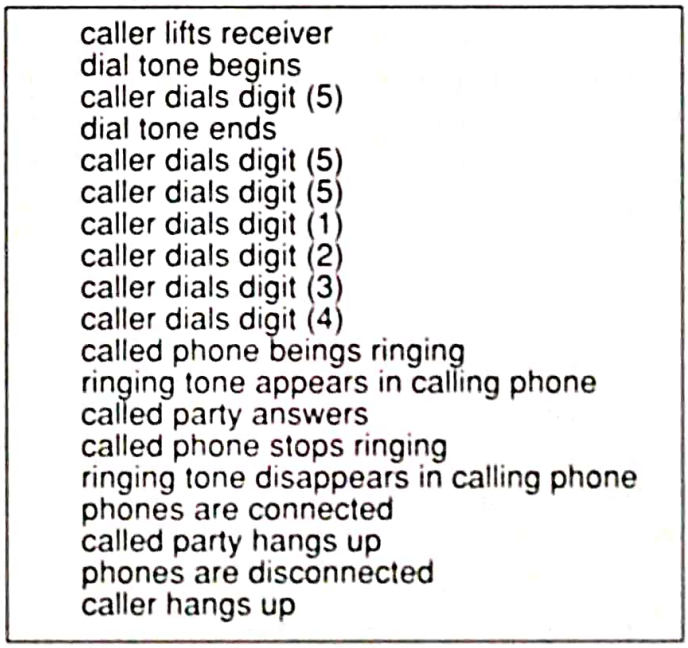

Figure 5.2 Scenario for phone call

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).

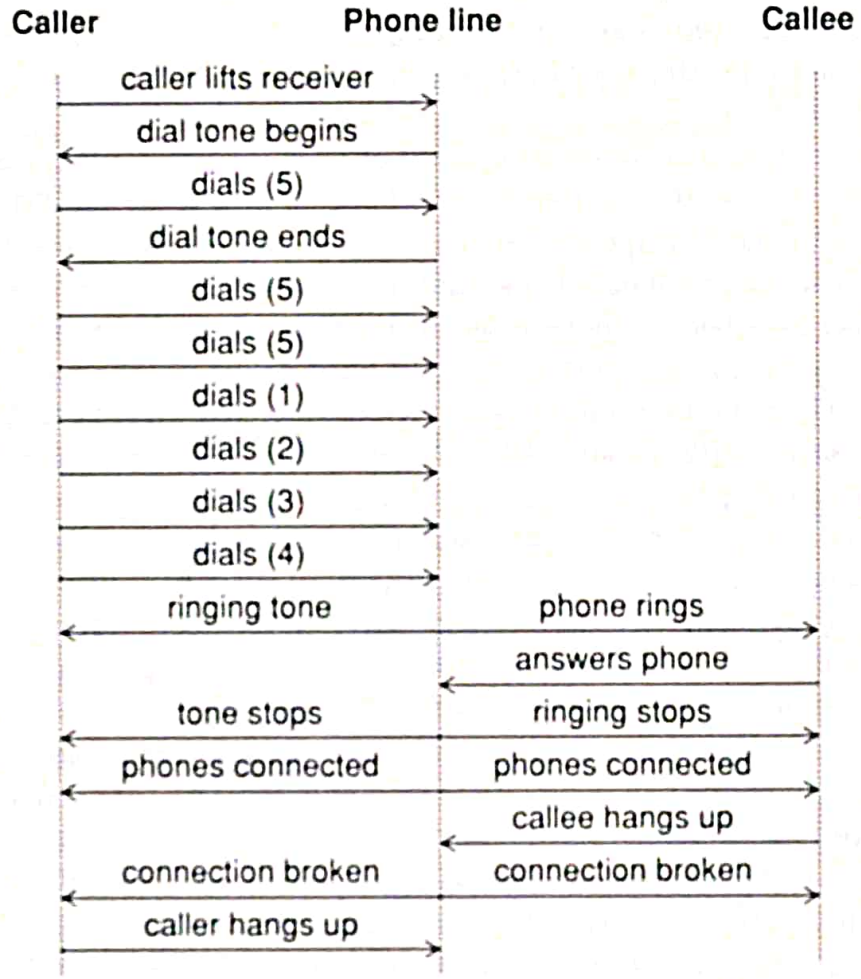

Figure 5.3 Event trace for phone call

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

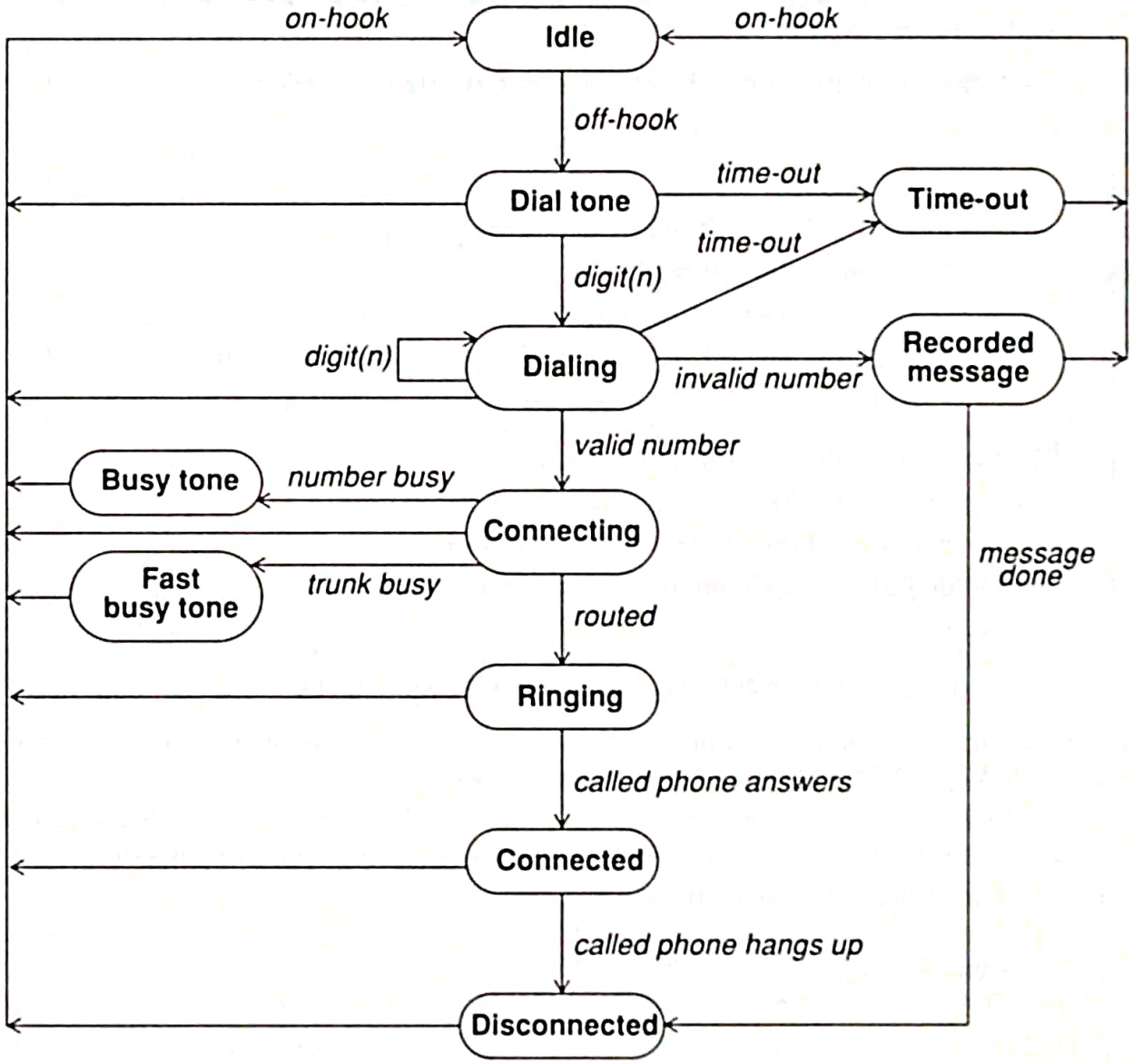

Figure 5.5 State diagram for phone line

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.

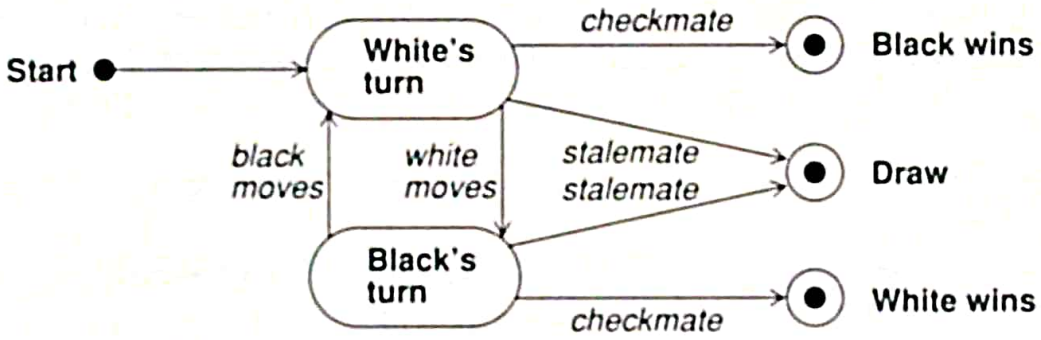

Figure 5.6 One-shot state diagram for chess game

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

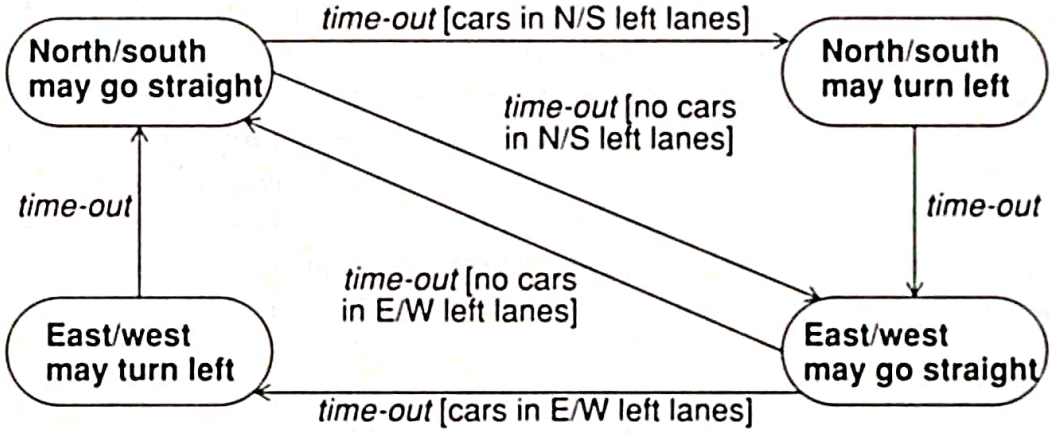

Figure 5.7 State diagram with guarded transitions

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.

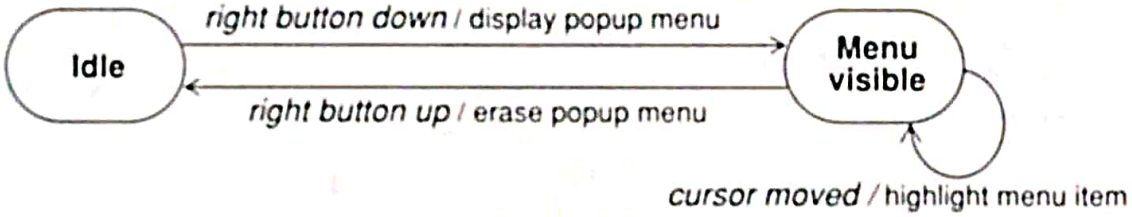

Figure 5.8 Actions for pop-up menu

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.

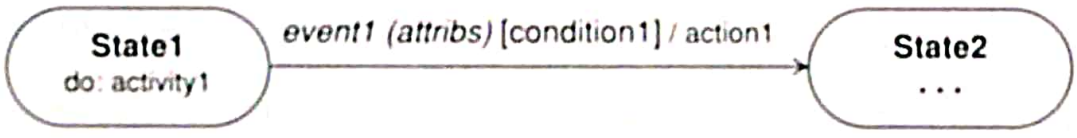

Figure 5.9 Summary of notation for unstructured state diagrams

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.

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

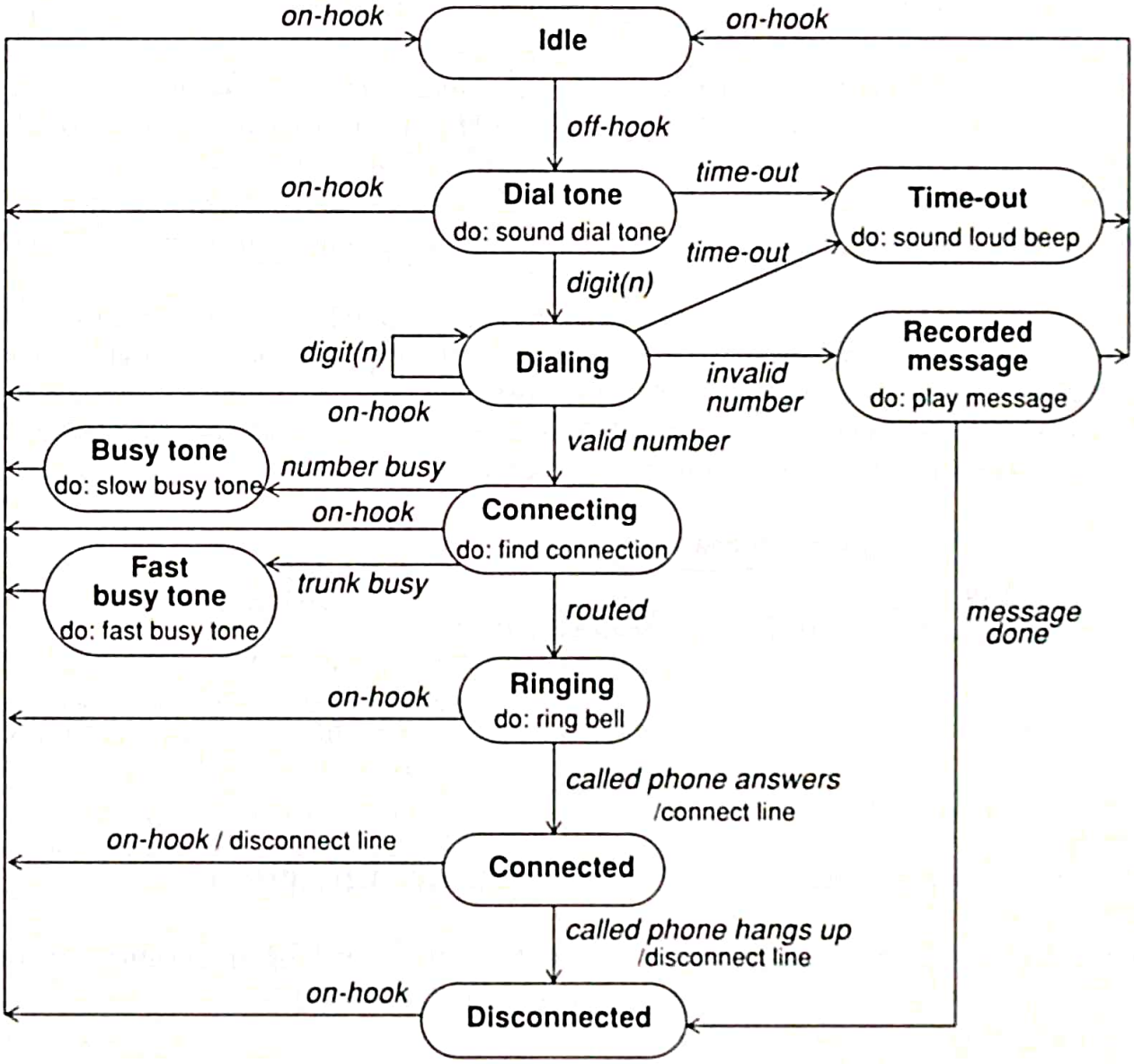

Figure 5.10 State diagram for phone line

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.

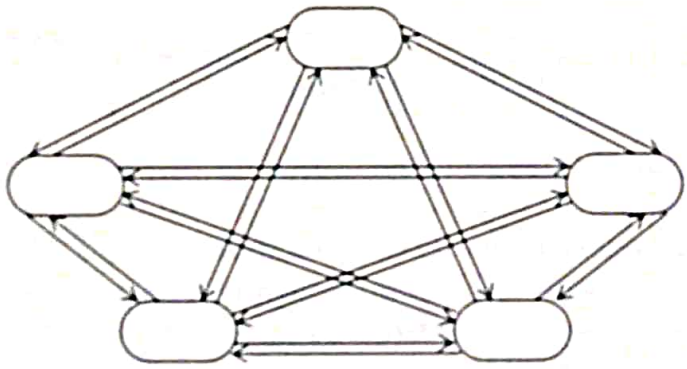

Figure 5.11 Combinatorial explosion of transitions in flat state diagram

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.

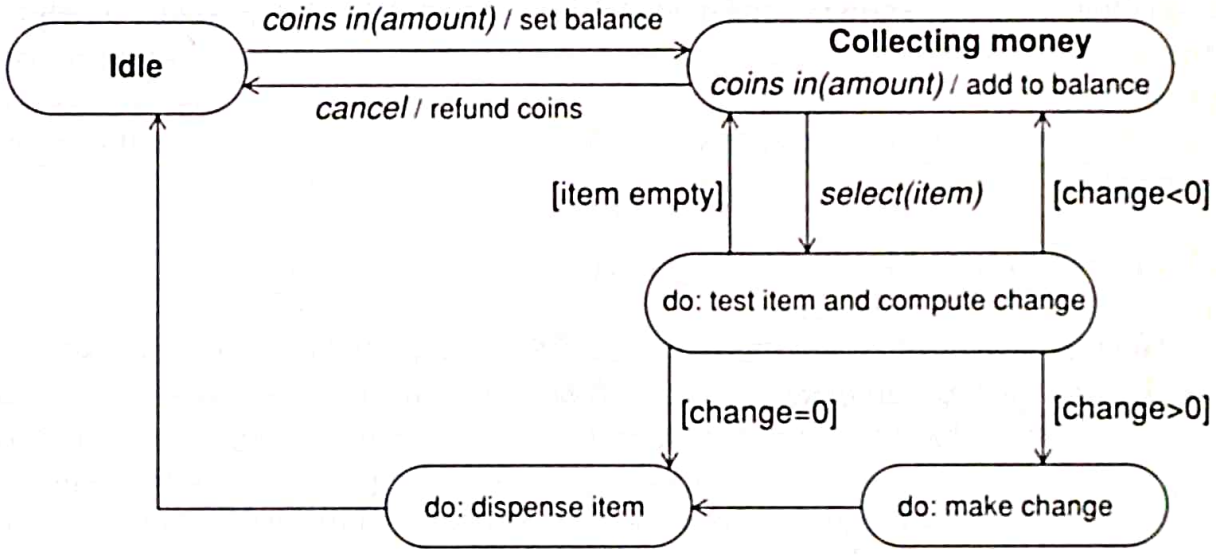

Figure 5.12 Vending machine model

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.

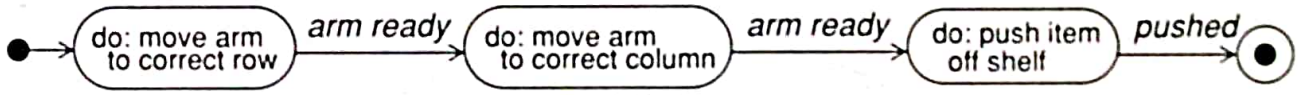

Figure 5.13 *Dispense item* activity of vending machine

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.

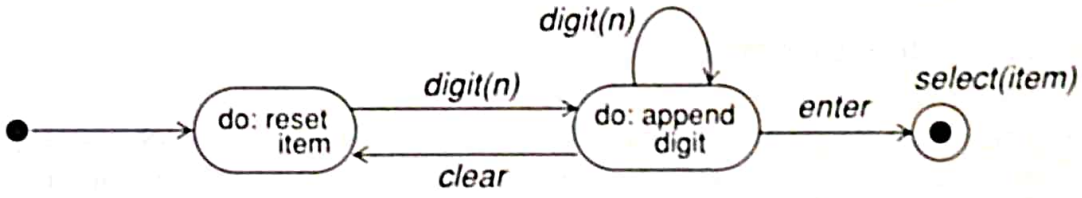

Figure 5.14 *Select item* transition of vending machine

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*.

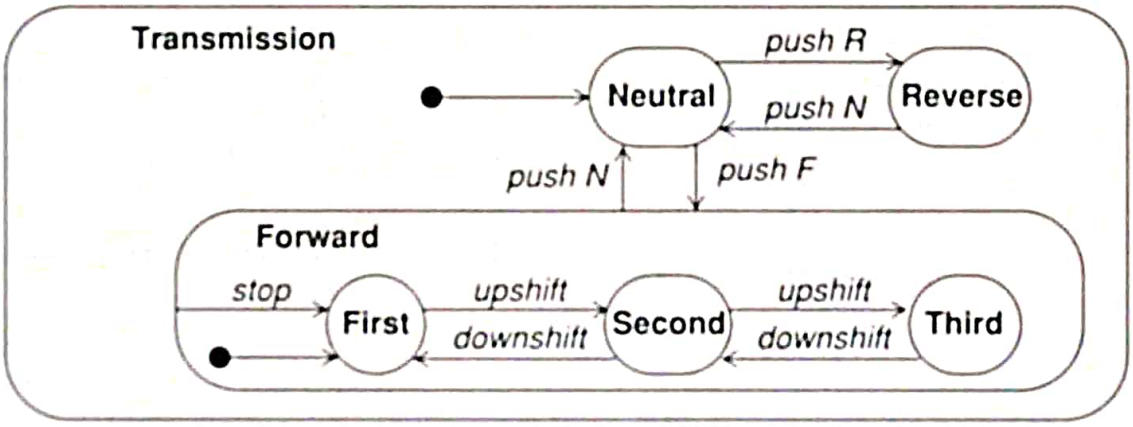

Figure 5.15 State diagram of car transmission with generalization

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*.

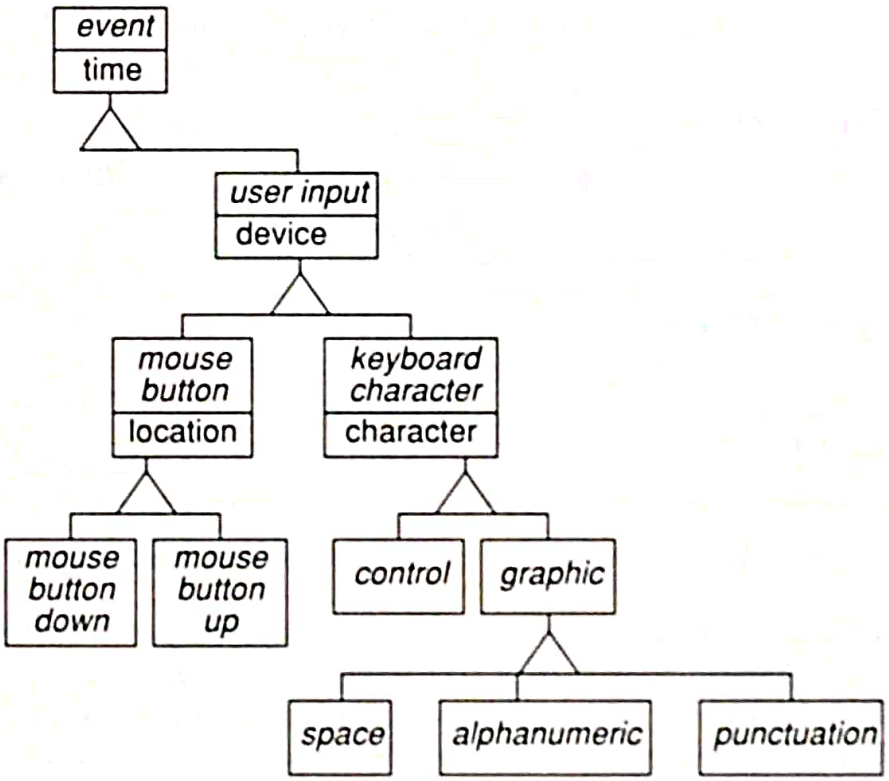

Figure 5.16 Partial event hierarchy for keyboard events

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.

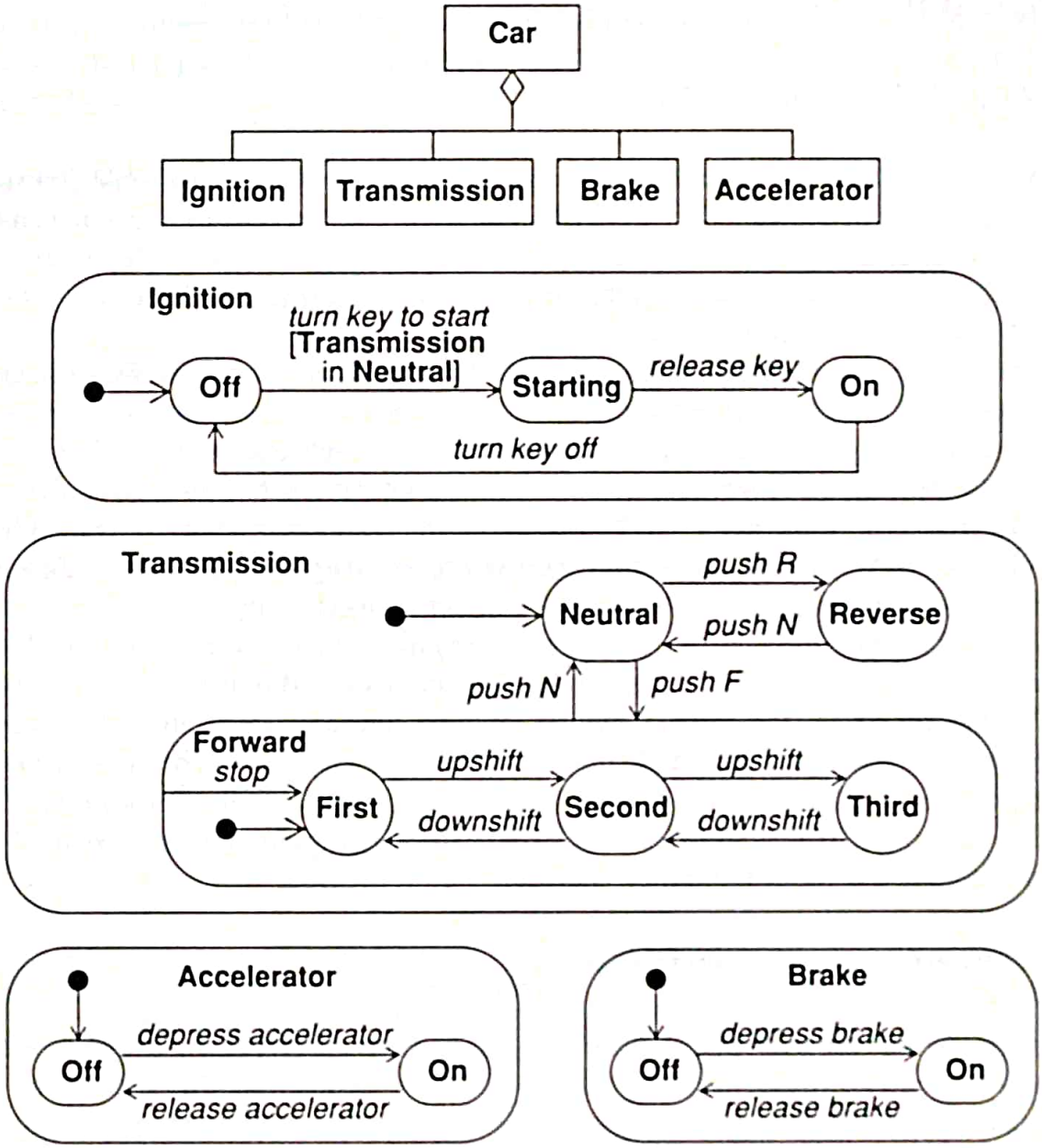

Figure 5.17 An aggregation and its concurrent state diagrams

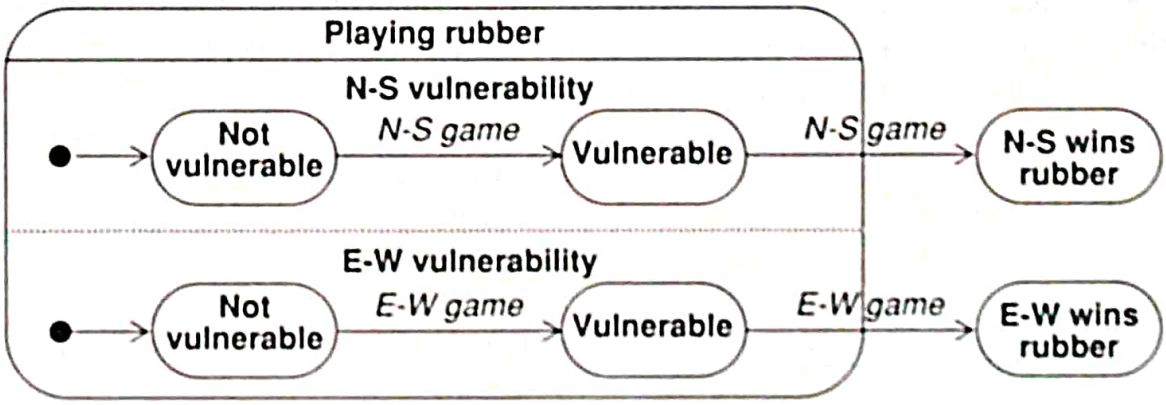

Figure 5.18 Bridge game with concurrent states

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.

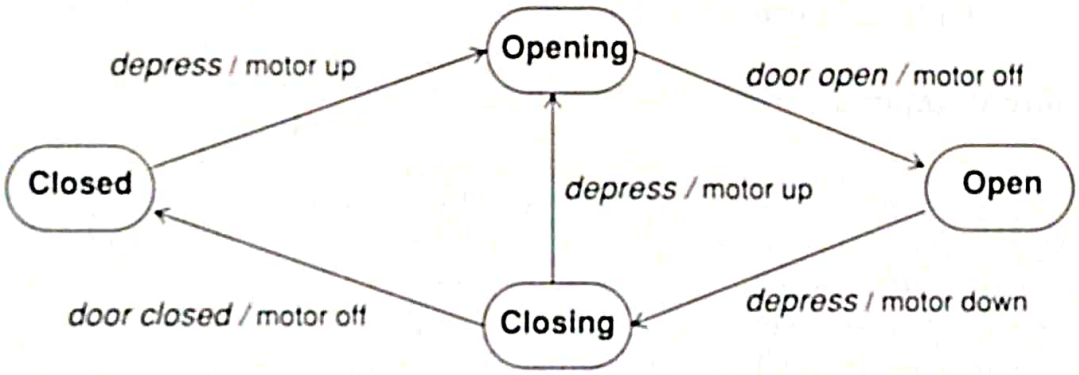

Figure 5.19 Actions on transitions

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.

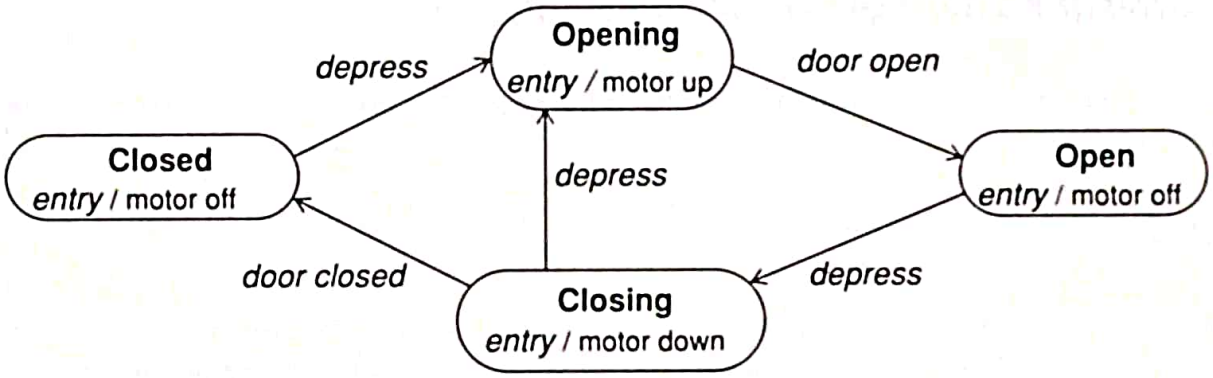

Figure 5.20 Actions on entry to states

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.

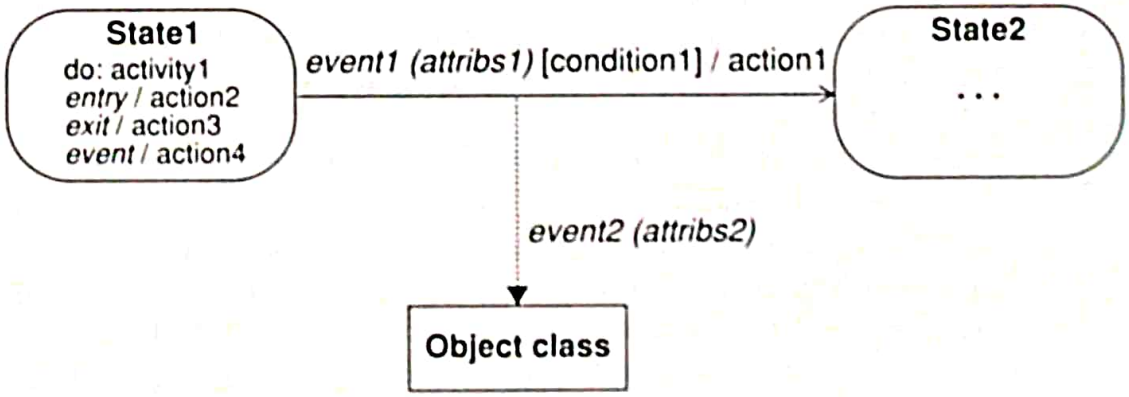

Figure 5.21 Summary of extended notation for state diagrams

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.

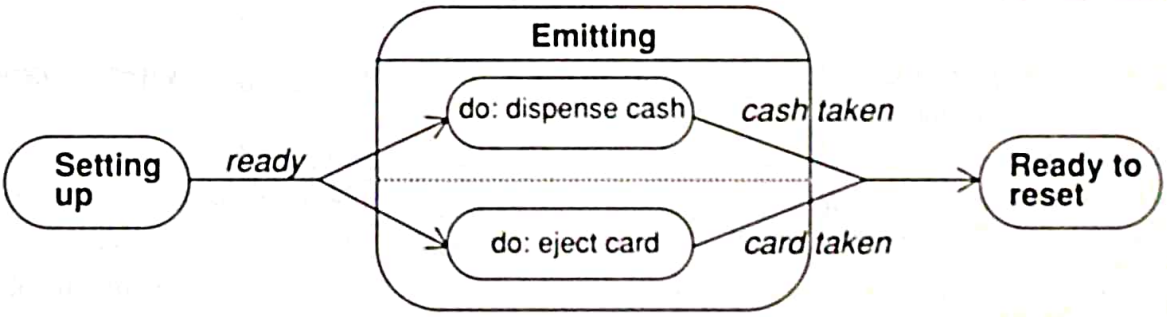

Figure 5.22 Synchronization of control

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.

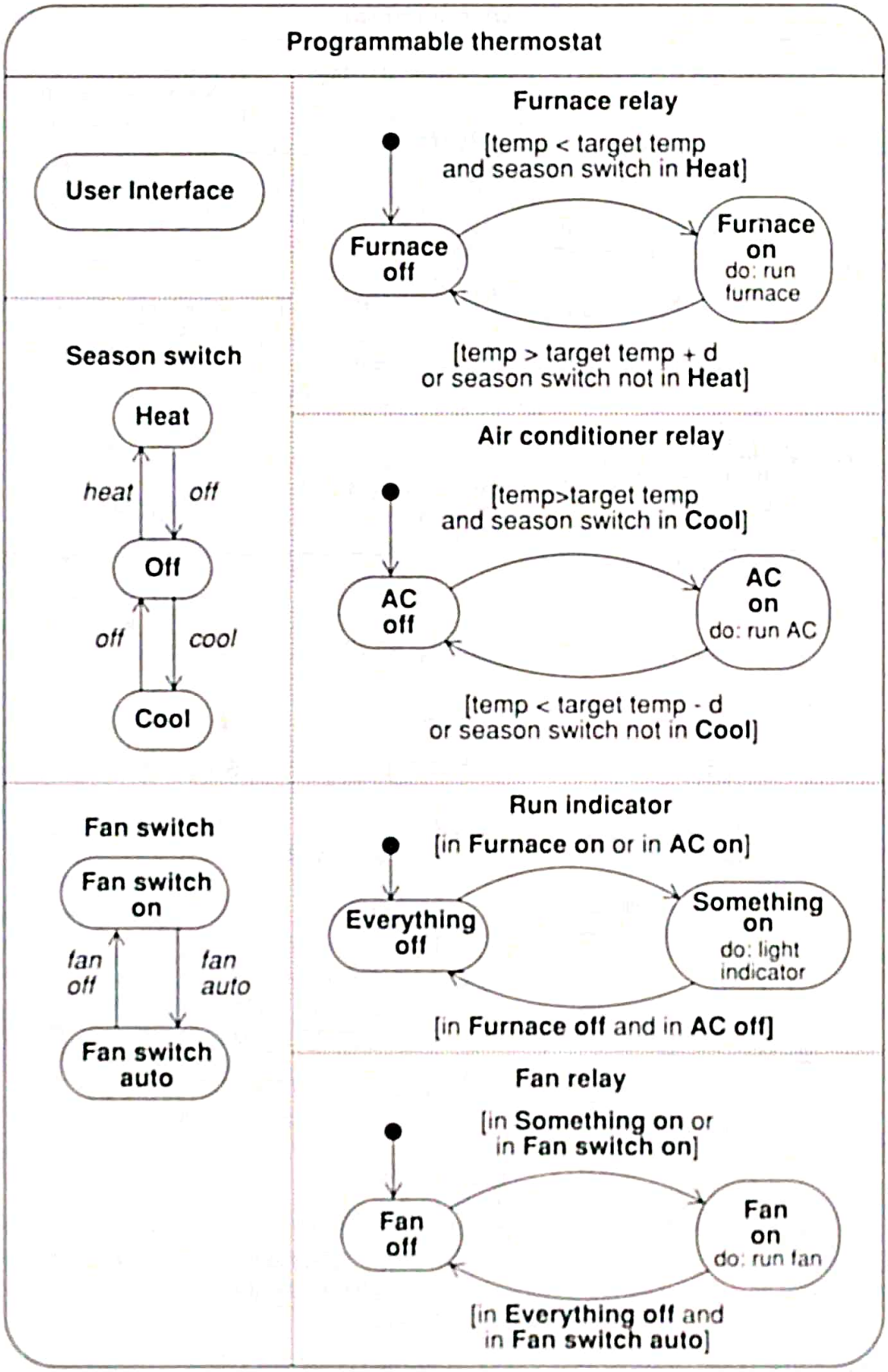

Figure 5.23 State diagram for programmable thermostat

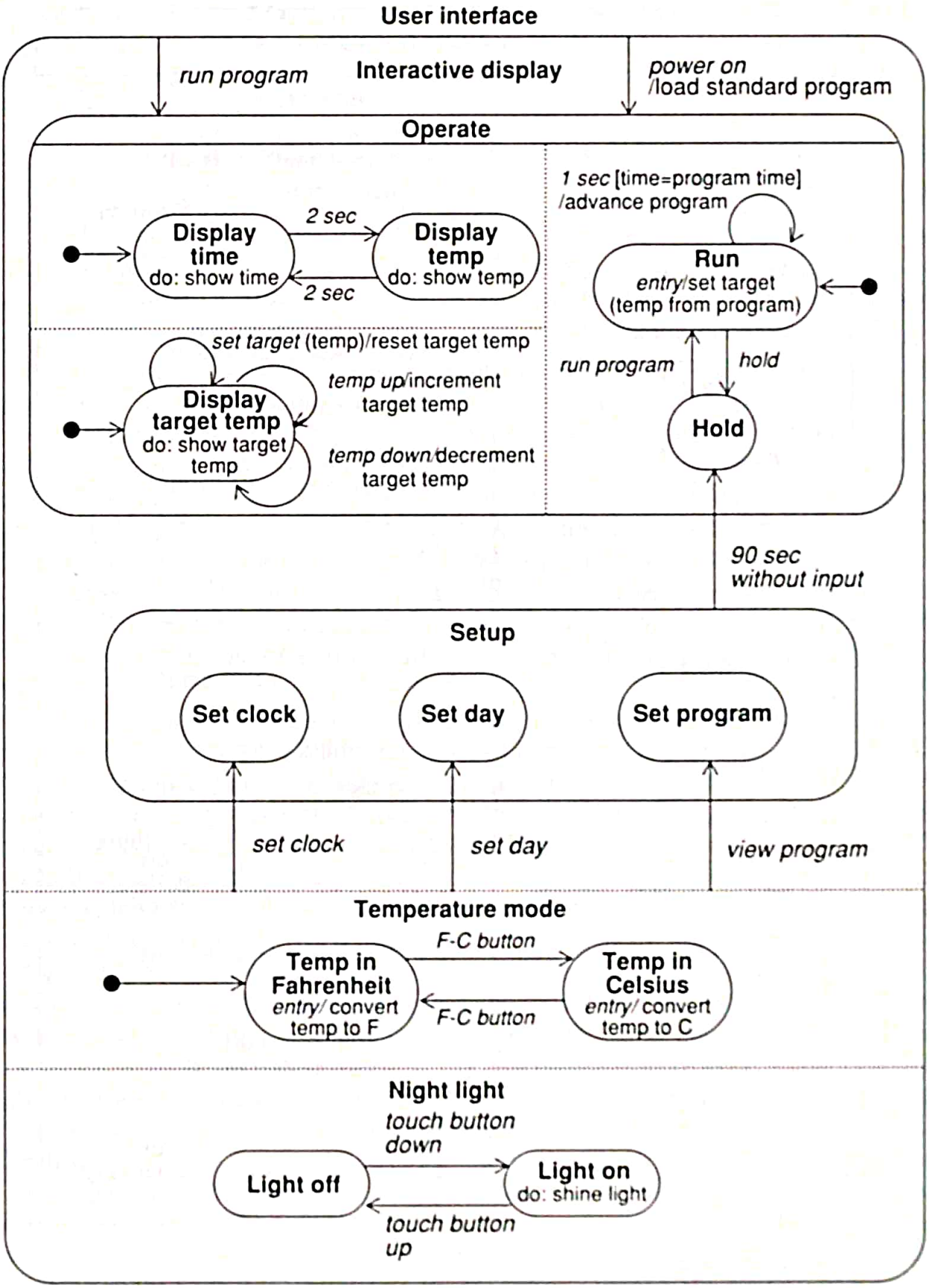

Figure 5.24 Subdiagram for thermostat user interface

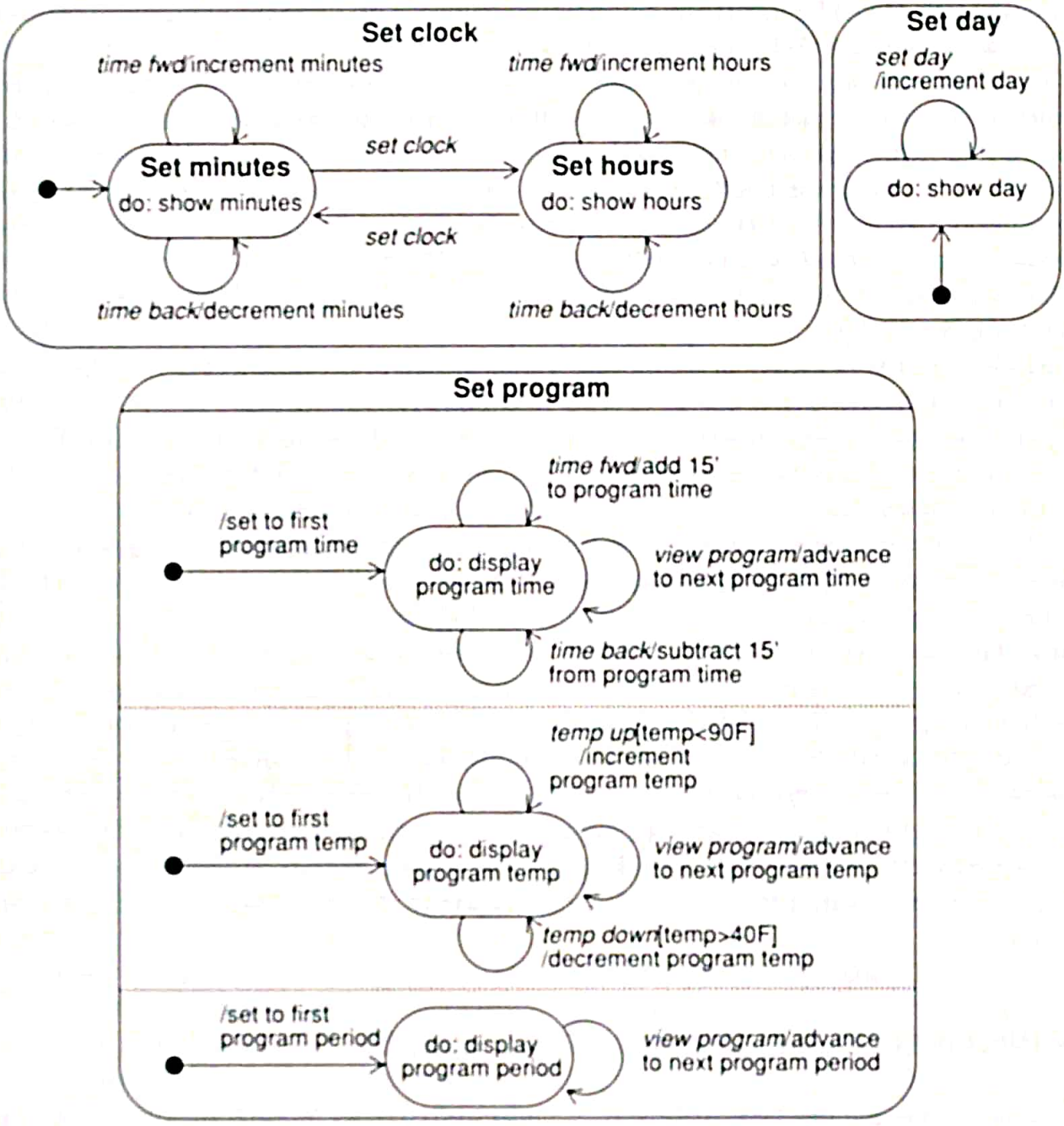

Figure 5.25 Subdiagrams for thermostat user interface setup

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.

6

Functional Modeling

The functional model describes computations within a system. The functional model is the third leg of the modeling tripod, in addition to the object model and the dynamic model. The functional model specifies what happens, the dynamic model specifies when it happens, and the object model specifies what it happens to.

The functional model shows how output values in a computation are derived from input values, without regard for the order in which the values are computed. The functional model consists of multiple data flow diagrams which show the flow of values from external inputs, through operations and internal data stores, to external outputs. The functional model also includes constraints among values within an object model. Data flow diagrams do not show control or object structure information; these belong to the dynamic and object models. We mainly follow the traditional exposition of data flow diagrams.

6.1 FUNCTIONAL MODELS

The functional model specifies the results of a computation without specifying how or when they are computed. The functional model specifies the meaning of the operations in the object model and the actions in the dynamic model, as well as any constraints in the object model. Noninteractive programs, such as compilers, have a trivial dynamic model; their purpose is to compute a function. The functional model is the main model for such programs, although the object model is important for any problem with nontrivial data structures. Many interactive programs also have a significant functional model. By contrast, databases often have a trivial functional model, since their purpose is to store and organize data, not to transform it.

A spreadsheet is a kind of functional model. In most cases, the values in the spreadsheet are trivial and cannot be structured further. The only interesting object structure is the cells in the spreadsheet itself. The purpose of the spreadsheet is to specify values in terms of other values.

A compiler is almost a pure computation. The input is the text of a program in a particular language; the output is an object file that implements the program in another language, often the machine language of a particular computer. The mechanics of compilation are irrelevant to the application.

The tax code is a large functional description. It specifies formulas for computing taxes based on income, expenses, donations, marital status, and so on. The tax code also defines objects (income, deductions) and contains dynamic information (when taxes are due, when estimated taxes are due, when income forms must be sent to employees). A set of tax forms and instructions is an algorithm implementing the functional model. Tax forms specify how to compute taxes based on a set of input values, such as income, expenses, deductions, and withholding. Note that tax forms only provide an algorithm for computing taxes; they do not define the actual tax due function. By contrast, the tax code usually defines the tax due function without specifying the algorithm for computing it. A taxpayer need not fill in the form in the exact sequence given in the instructions to get the correct answer.

6.2 DATA FLOW DIAGRAMS

The functional model consists of multiple data flow diagrams which specify the meaning of operations and constraints. A data flow diagram (DFD) shows the functional relationships of the values computed by a system, including input values, output values, and internal data stores. A data flow diagram is a graph showing the flow of data values from their sources in objects through *processes* that transform them to their destinations in other objects. A data flow diagram does not show control information, such as the time at which processes are executed or decisions among alternate data paths; this information belongs to the dynamic model. (Some authors do include control information in DFDs, primarily to show everything on one diagram, but we have broken out the control information into a separate diagram, the state diagram.) A data flow diagram does not show the organization of values into objects; this information belongs to the object model.

A data flow diagram contains *processes* that transform data, *data flows* that move data, *actor* objects that produce and consume data, and *data store* objects that store data passively. Figure 6.1 shows a data flow diagram for the display of an icon on a windowing system. The icon name and location are inputs to the diagram from an unspecified source. The icon is expanded to vectors in the application coordinate system using existing icon definitions. The vectors are clipped to the size of the window, then offset by the location of the window on the screen, to obtain vectors in the screen coordinate system. Finally the vectors are converted to pixel operations that are sent to the screen buffer for display. The data flow diagram shows the sequence of transformations performed, as well as the external values and objects that affect the computation.

6.2.1 Processes

A *process* transforms data values. The lowest-level processes are pure functions without side effects. Typical functions include the sum of two numbers, the finance charge on a set of

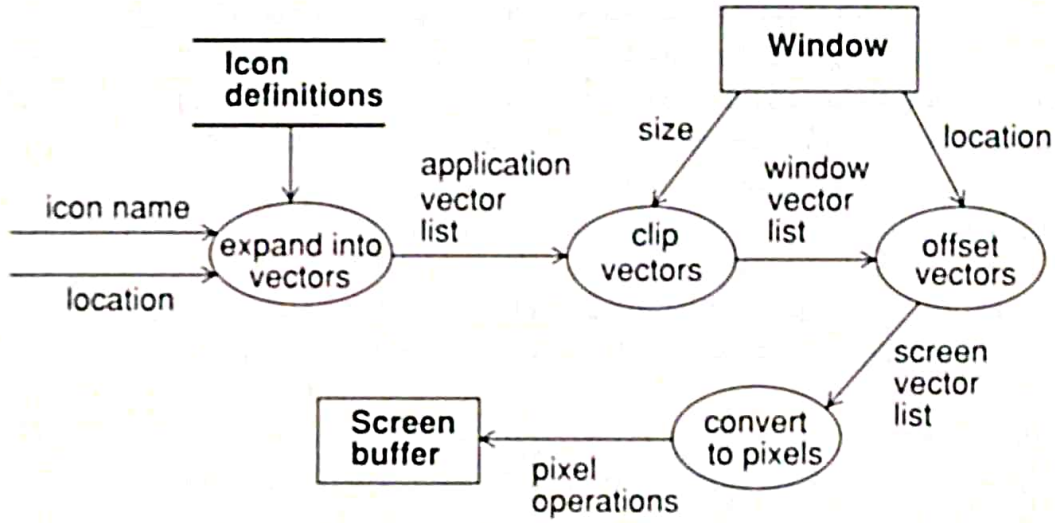

Figure 6.1 Data flow diagram for windowed graphics display

credit card transactions, and the spline through a list of points. An entire data flow graph is a high-level process. A process may have side effects if it contains nonfunctional components, such as data stores or external objects. The functional model does not uniquely specify the results of a process with side effects. The functional model only indicates the possible functional paths; it does not show which path will actually occur. The results of such a process depend on the behavior of the system, as specified by the dynamic model. Examples of nonfunctional processes include reading and writing files, a voice recognition algorithm that learns from experience, and the display of images within a workstation windowing system.

A process is drawn as an ellipse containing a description of the transformation, usually its name. Each process has a fixed number of input and output data arrows, each of which carries a value of a given type. The inputs and outputs can be labeled to show their role in the computation, but often the type of value on the data flow is sufficient. Figure 6.2 shows two processes. Note that a process can have more than one output. The *display icon* process represents the entire data flow diagram of Figure 6.1 at a higher level of abstraction.

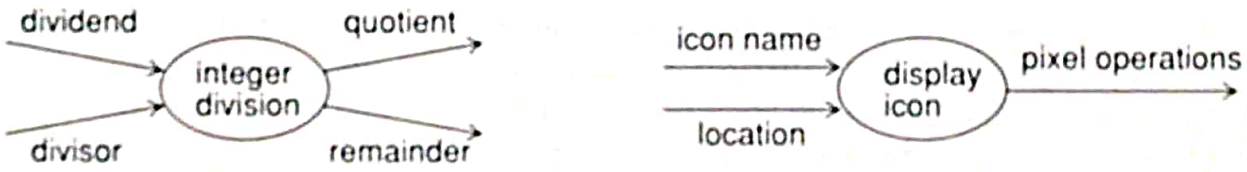

Figure 6.2 Processes

The diagram only shows the pattern of inputs and outputs. The computation of output values from input values must also be specified. A high-level process can be expanded into an entire data flow diagram, much as a subroutine can be expanded into lower-level subroutines. Eventually the recursion must stop, and the atomic processes must be described directly, in natural language, mathematical equations, or by some other means. For example, "integer division" could be defined mathematically and "display icon" would be defined in terms of Figure 6.1. Frequently the atomic processes are trivial and simply access a value from an object, for example.

Processes are implemented as methods (or method fragments) of operations on object classes. The target object is usually one of the input flows, especially if the same class of object is also an output flow. In some cases, however, the target object is implicit. For example, in Figure 6.2, the target of *display icon* is the window that receives the pixel operations.

6.2.2 Data Flows

A *data flow* connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. The value is not changed by the data flow.

A data flow is drawn as an arrow between the producer and the consumer of the data value. The arrow is labeled with a description of the data, usually its name or type. The same value can be sent to several places; this is indicated by a fork with several arrows emerging from it. The output arrows are unlabeled because they represent the same value as the input. Figure 6.3 shows some data flows.

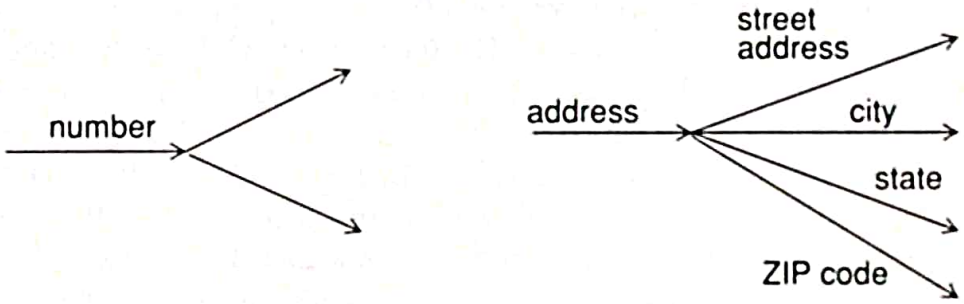

Figure 6.3 Data flows to copy a value and split an aggregate value

Sometimes an aggregate data value is split into its components, each of which goes to a different process. This is shown by a fork in the path in which each outgoing arrow is labeled with the name of its component. The combination of several components into an aggregate value is just the opposite.

Each data flow represents a value at some point in the computation. The data flows internal to the diagram represent intermediate values within a computation and do not necessarily have any significance in the real world.

Flows on the boundary of a data flow diagram are its inputs and outputs. These flows may be unconnected (if the diagram is a fragment of a complete system), or they may be connected to objects. The inputs of Figure 6.1 are *icon name* and *location*; their sources must be specified in the larger context in which the diagram is used. The outputs of Figure 6.1 are *pixel operations*, which are sent to the screen buffer object. The same inputs and outputs appear in the bottom part of Figure 6.2, in which the entire data flow diagram of Figure 6.1 has been abstracted into a process.

6.2.3 Actors

An *actor* is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph. In a sense, the actors

lie on the boundary of the data flow graph but terminate the flow of data as sources and sinks of data, and so are sometimes called *terminators*. Examples of actors include the user of a program, a thermostat, and a motor under computer control. The actions of the actors are outside the scope of the data flow diagram but should be part of the dynamic model.

An actor is drawn as a rectangle to show that it is an object. Arrows between the actor and the diagram are inputs and outputs of the diagram. The screen buffer in Figure 6.1 is an actor that consumes pixel operations.

6.2.4 Data Stores

A *data store* is a passive object within a data flow diagram that stores data for later access. Unlike an actor, a data store does not generate any operations on its own but merely responds to requests to store and access data. A data store allows values to be accessed in a different order than they are generated. Aggregate data stores, such as lists and tables, provide access of data by insertion order or index keys. Sample data stores include a database of airline seat reservations, a bank account, and a list of temperature readings over the past day.

A data store is drawn as a pair of parallel lines containing the name of the store. Input arrows indicate information or operations that modify the stored data; this includes adding elements, modifying values, or deleting elements. Output arrows indicate information retrieved from the store. This includes retrieving the entire value or some component of it. The actual structure of the object must be described in the object model, together with a description of the update and access operations permitted.

Figure 6.4a shows a data store for temperature readings. Every hour a new temperature reading enters the store. At the end of the day, the maximum and minimum reading are retrieved from the store. In addition to introducing delays in the use of data, data stores permit many pieces of data to be accumulated and then used at once.

Figure 6.4b shows a data store for a bank account. The double-headed arrow indicates that *balance* is both an input and an output of the subtraction operation. This could be drawn with two separate arrows, but accessing and updating a value in a data store is a common operation.

Figure 6.4c shows a price list for items. Input to the store consists of pairs of item name and cost values. Later an item is given, and the corresponding cost is found. The unlabeled arrow from the data store to the process indicates that the entire price list is an input to the selection operation. Note that the item name is not an input to the data store during the selection operation because it does not modify the store but merely supplies input to the selection process.

Figure 6.4d shows the periodic table being accessed to find the atomic weight of an element. Obviously the properties of chemical elements are constant and not a variable of the program. It is convenient to represent the operation as a simple access of a *constant data store* object. Such a data store has no inputs.

Both actors and data stores are objects. We distinguish them because their behavior and usage is generally different, although in an object-oriented language they might both be implemented as objects. On the other hand, a data store might be implemented as a file and an actor as an external device. Some data flows are also objects, although in many cases they

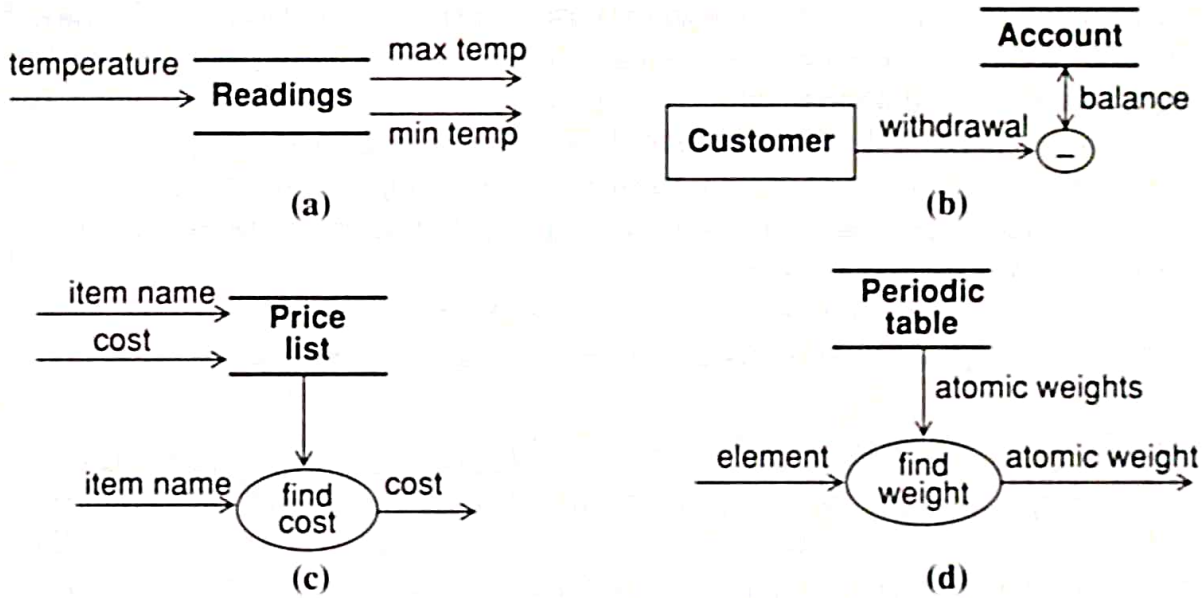

Figure 6.4 Data stores

are pure values, such as integers, which lack individual identity. (In an object-oriented language, however, objects and pure values are often implemented the same.)

There is a difference between viewing an object as a single value and as a data store containing many values. In Figure 6.5, the customer name selects an account from the bank. The result of this operation is the account object itself, which is then used as a data store in the update operation. A data flow that generates an object used as the target of another operation is indicated by a hollow triangle at the end of the data flow. In contrast, the update operation modifies the balance in the account object, as indicated by the small arrowhead. The hollow triangle indicates a data flow value that subsequently is treated as an object, usually a data store. (This is a new construct that we have introduced. Traditional data flow notation does not adequately represent the dynamic creation or selection of an object for later use in the diagram as an aggregate.)

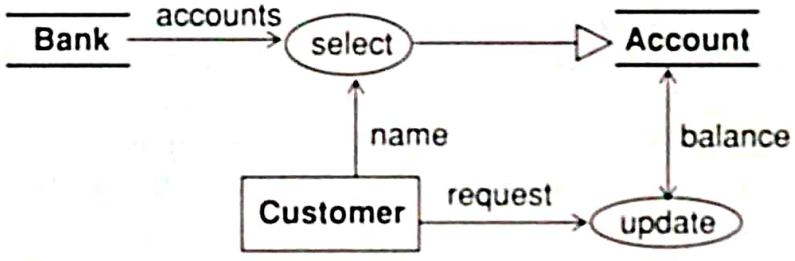

Figure 6.5 Selection with an object as result

Figure 6.6 shows the creation of a new account in a bank. The result of the *create account* process is a new account, which is stored in the bank. The customer's name and deposit are stored in the account. The account number from the new account is given to the customer. In this example, the account object is viewed both as a data value (stored in the bank) and as a data store (used to store and retrieve values).

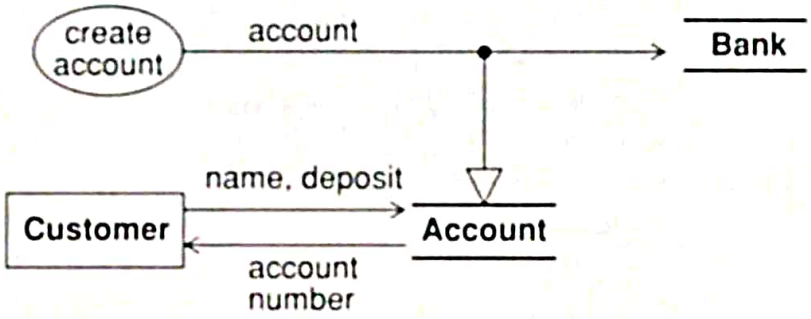

Figure 6.6 Creation of a new object

6.2.5 Nested Data Flow Diagrams

A data flow diagram is particularly useful for showing the high-level functionality of a system and its breakdown into smaller functional units. A process can be expanded into another data flow diagram. Each input and output of the process is an input or output of the new diagram. The new diagram may have data stores that are not shown in the higher-level diagram. The *display icon* process of Figure 6.2 corresponds to the data flow diagram of Figure 6.1. Diagrams can be nested to an arbitrary depth, and the entire set of nested diagrams forms a tree. Nesting of a data flow diagram permits each level to be coherent and understandable yet the overall functionality can be arbitrarily complex. A diagram that references itself represents a recursive computation. (The nesting of diagrams has also been called *leveling*, since the diagrams are organized into different levels.)

Eventually the nesting of diagrams terminates with simple functions. These functions must be specified as operations, to be explained in Section 6.3.

6.2.6 Control Flows

A data flow diagram shows all possible computation paths for values; it does not show which paths are executed and in what order. Decisions and sequencing are control issues that are part of the dynamic model. A decision affects whether one or more functions are even performed, rather than supplying a value to the functions. Even though the functions do not have input values from these decision functions, it is sometimes useful to include them in the functional model so that they are not forgotten and so their data dependencies can be shown. This is done by including *control flows* in the data flow diagram.

A control flow is a Boolean value that affects whether a process is evaluated. The control flow is not an input value to the process itself. A control flow is shown by a dotted line from a process producing a Boolean value to the process being controlled.

Figure 6.7 shows a data flow diagram for a withdrawal from a bank account. The customer supplies a password and an amount. The withdrawal occurs only if the password is successfully verified. The update process could be expanded with a similar control flow to guard against overdrafts.

Control flows can occasionally be useful, but they duplicate information in the dynamic model and should be used sparingly.

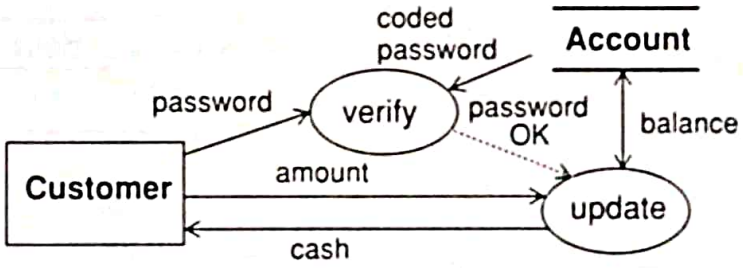

Figure 6.7 Control flow

6.3 SPECIFYING OPERATIONS

Processes in data flow diagrams must eventually be implemented as operations on objects. Each bottom-level, atomic process is an operation. Higher-level processes may also be considered operations, although an implementation may be organized differently from the data flow diagram it represents because of optimization. Each operation may be specified in various ways, including the following:

- mathematical functions, such as trigonometric functions;
- table of input and output values (enumeration) for small finite sets;
- equations specifying output in terms of input;
- pre- and post-conditions (axiomatic definition);
- decision tables;
- pseudocode;
- natural language.

Specification of an operation includes a signature and a transformation. The signature defines the interface to the operation: the arguments it requires (number, order, and types) and the values it returns (number, order, and types). The operation is usually listed in the object model to show the pattern of inheritance; the signature of all methods implementing an operation must match. The transformation defines the effect of an operation: the output values as functions of the input values and the side effects of the operation on its operand objects.

The external specification of an operation only describes changes visible outside the operation. During the implementation of an operation, internal values may be created for convenience or optimization. Some values may even be part of the internal state of an object. For example, a sorted list of values can be implemented using various data structures, such as a linear list or a balanced tree, whose internal organization can be freely changed provided it does not change the external ordering of the list. Such internal details are private to an operation (and possibly to an object class) and do not appear in its external specification. The purpose of the specification is to indicate what an operation must do logically, not how it must be implemented. Therefore the state of the object itself must be divided into externally visible information and private internal information. Changes to the internal state of an object that are not externally visible do not change the value of the object.

Access operations are operations that read or write attributes or links of an object. It is unnecessary to list or specify access operations during analysis, since they are trivial. During design, it is necessary to note which access operations will be public and which will be private to the object class. (See Section 1.2.2 for an explanation of analysis and design.) The reason for restricting access is not for reasons of logical correctness but rather to encapsulate classes for protection against bugs and to permit modifications to the implementation in the future. Access operations are derived directly from the attributes and associations of a class in the object model.

Nontrivial operations can be divided into three categories: queries, actions, and activities. A *query* is an operation that has no side effects on the externally visible state of any object; it is a pure function. A query with no parameters (except for the target object) is a *derived attribute*; it has the form (although not necessarily the implementation) of an attribute. For example, if a point is specified in Cartesian coordinates, then the radius and angle are derived attributes. In the object model, query operations can be grouped with attributes, but their derived status should be indicated because they do not contribute additional information to the state of the object. In many cases, the choice of certain attributes as base attributes and others as derived attributes is arbitrary. For example, a point may be expressed in both Cartesian and polar coordinates; neither is more correct. Because query operations have no external effects, they are less important in analyzing and designing a system than base attributes and actions. They can often be specified with equations written in terms of other attributes and do not require a control component. Query operations are derived from paths in the object model or by repackaging data from the object model.

An *action* is a transformation that has side effects on the target object or other objects in the system reachable from the target object. An action has no duration in time; it is logically instantaneous (although any actual implementation will take some time, of course). Because the state of an object is defined by its attributes and links, all actions must be definable in terms of updates to base attributes and links. An action can be defined in terms of the state of the system before and after the action; a control component is unnecessary. For example, the action of scaling a window on a workstation involves scaling the window boundary and all of the contents of the window by a fixed factor. The order in which the scaling is performed is irrelevant to the specification of the action; only the final result matters. Actions are usually derived from processes in the functional model.

Actions can be described in various ways, including mathematical equations, decision trees, decision tables, enumeration of all possible inputs, predicate calculus, and natural language. It is important that a specification be clear and unambiguous, not that it be formal. Figure 6.8 shows the specification for a telephone switcher connecting a call. There are several elements of a specification: the function name, the inputs and outputs, the transformations to values, and the constraints that must be observed. Note that no algorithm for determining connections is given. The specification is informal and still ambiguous. For example, the topology of the network must be specified in more detail. Nevertheless, this will do for an initial top-level specification.

One way of specifying an action is to give an algorithm (English, pseudocode, or actual code) for computing it. Frequently a simple but inefficient algorithm for a function is easy to define. This does not imply that the program must use the same algorithm, only that the results be identical, although proving that two algorithms yield the same result can be diffi-

Function: connect call

Inputs: phone line, number dialed, current settings of switches

Outputs: new settings of switches, connection status

Transformation: Connect the calling phone to the dialed phone by closing connections in the switcher, observing the following constraints:

Constraints: Only two lines are a time may be connected on any one circuit.

Previous connections must not be disturbed.

If the called line is already in use, then no switches are closed, and the status is reported as busy.

If a connection is impossible because too many switches are in use, then no switches are closed, and the status is reported as switcher busy.

Figure 6.8 Action for telephone switcher connection

cult. Some functions can be specified in ways that do not provide a basis for an algorithm, even an inefficient one. For example, the inverse of a matrix A is defined as "that matrix B such that A times B yields the identity matrix." Matrix multiplication is easily defined, but deriving an algorithm to compute the inverse requires a considerable amount of linear algebra. Supplying an algorithm is part of design.

An *activity* is an operation to or by an object that has duration in time, as opposed to queries and actions, which are considered as instantaneous (logically, if not actually). An activity inherently has side effects because of its extension in time. Activities only make sense for actors, objects that generate operations on their own, because passive objects are mere data repositories. An operating system demon, such as an output spooler, is considered an actor because it has an active role in controlling the flow of information. The details of an activity are specified by the dynamic model as well as the functional model and cannot be considered just as a transformation. In most cases, an activity corresponds to a state diagram in the dynamic model.

6.4 CONSTRAINTS

A *constraint* shows the relationship between two objects at the same time (such as frequency and wavelength) or between different values of the same object at different times (such as the number of outstanding shares of the mutual fund). A constraint may be expressed as a total function (one value is completely specified by another) or a partial function (one value is restricted, but not completed specified, by another). For example, a coordinate transformation might specify that the scale factor for the x-coordinate and the y-coordinate will be equal; this constraint totally defines one value in terms of the other. The Second Law of Ther-

modynamics expresses a partial constraint; it states that the entropy (disorder) of the Universe can never decrease.

Constraints can appear in each of the kinds of model. Object constraints specify that some objects depend entirely or partially on other objects. Dynamic constraints specify relationships among the states or events of different objects. Functional constraints specify restrictions on operations, such as the scaling transformation described above.

A constraint between values of an object over time is often called an *invariant*. Conservation laws in physics are invariants: The total energy, or charge, or angular momentum of a system remains constant. Invariants are useful in specifying the behavior of operations.

6.5 A SAMPLE FUNCTIONAL MODEL

In this section, we describe the functional model for a flight simulator. The simulator is responsible for handling pilot input controls, computing the motion of the airplane, computing and displaying the outside view from the cockpit window, and displaying the cockpit gauges. The simulator is intended to be an accurate but simplified model of flying an airplane, ignoring some of the smaller effects and making some simplifying assumptions. For example, we omit the rudder, under the assumption that it is held so as to keep the plane pointing in the direction of motion. Figure 6.9 shows the top-level data flow diagram for the flight simulator. There are two input actors: the *Pilot*, who operates the airplane controls, and the *Weather*, which varies according to some specified pattern. There is one output actor: the *Screen*, which displays the pilot's view. There are two read-only data stores: the *Terrain database*, which specifies the geometry of the surrounding terrain as a set of colored polygonal surfaces, and the *Cockpit database*, which specifies the shape and location of the cockpit viewport and the locations of the various gauges. There are three internal data stores: *Spatial parameters*, which holds the 3-D position, velocity, orientation, and rotation of the plane; *Fuel*, which holds the amount of fuel remaining; and *Weight*, which holds the total weight of the plane (consumption of fuel causes the weight to decrease). The initialization of the internal data stores is necessary but is not shown on the data flow diagram.

The processes in the diagram can be divided into three kinds: handling controls, motion computation, and display generation. The control handling processes are *adjust controls*, which transforms the position of the pilot's controls (such as joysticks) into positions of the airplane control surfaces and engine speed; *consume fuel*, which computes fuel consumption as a function of engine speed; and *compute weight*, which computes the weight of the airplane as the sum of the base weight and the weight of the remaining fuel. Process *adjust controls* is expanded on Figure 6.10, where it can be seen as comprising three distinct controls: the elevator, the ailerons, and the throttle. There is no need to expand these processes further, as they can be described by input-output functions (we do not attempt to specify them here).

The motion computation processes are *compute forces*, which computes the various forces and torques on the plane and sums them to determine the net acceleration and the rotational torques, and *integrate motion*, which integrates the differential equations of motion. Process *compute forces* incorporates both geometrical and aeronautical computations. It is expanded on Figure 6.11. Net force is computed as the vector sum of drag, lift, thrust, and

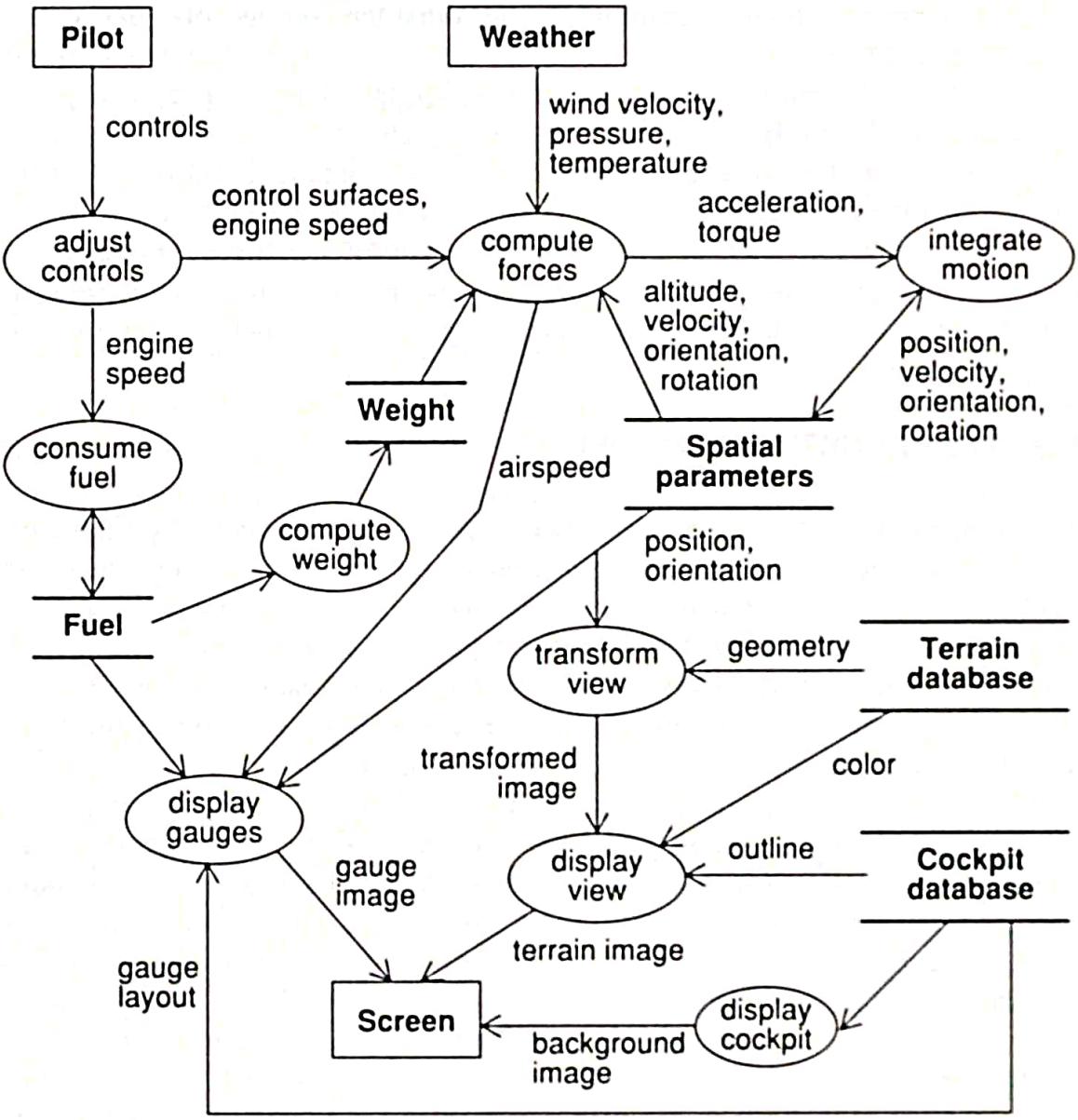

Figure 6.9 Functional model of flight simulator

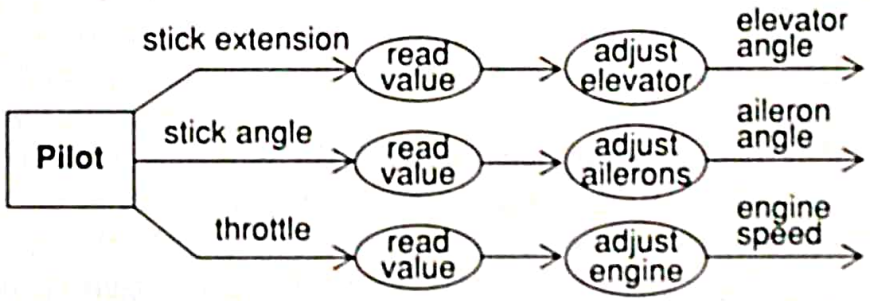

Figure 6.10 Expansion of *adjust controls* process

weight. These forces in turn depend on intermediate parameters, such as airspeed, angle of attack, and air density. The aerodynamic calculations must be made relative to the air mass, so the wind velocity is subtracted from the plane's velocity to give the airspeed relative to

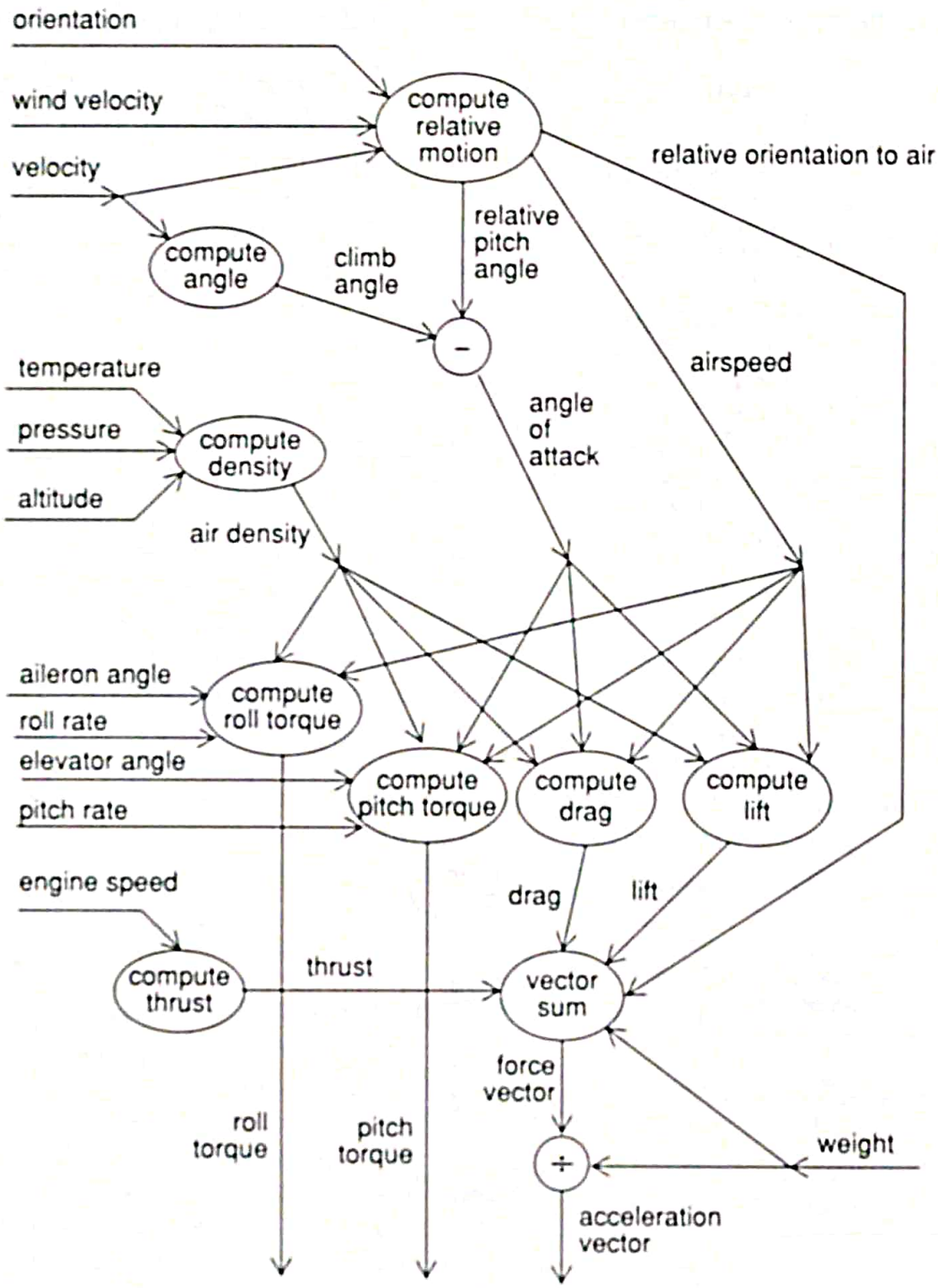

Figure 6.11 Expansion of *compute forces* processes

the air mass; the orientation of the plane must be transformed also. Air density is also computed and used in subsequent processes. The intermediate parameters are computed in terms of data store parameters, such as airplane velocity, orientation, rotation rates roll rate and pitch rate, and altitude, obtained from *Spatial parameters*; wind velocity, temperature, and pressure, obtained from *Weather*; weight, obtained from *Weight*; and in terms of output data flows from other processes, such as elevator angle, aileron angle, and engine speed, obtained

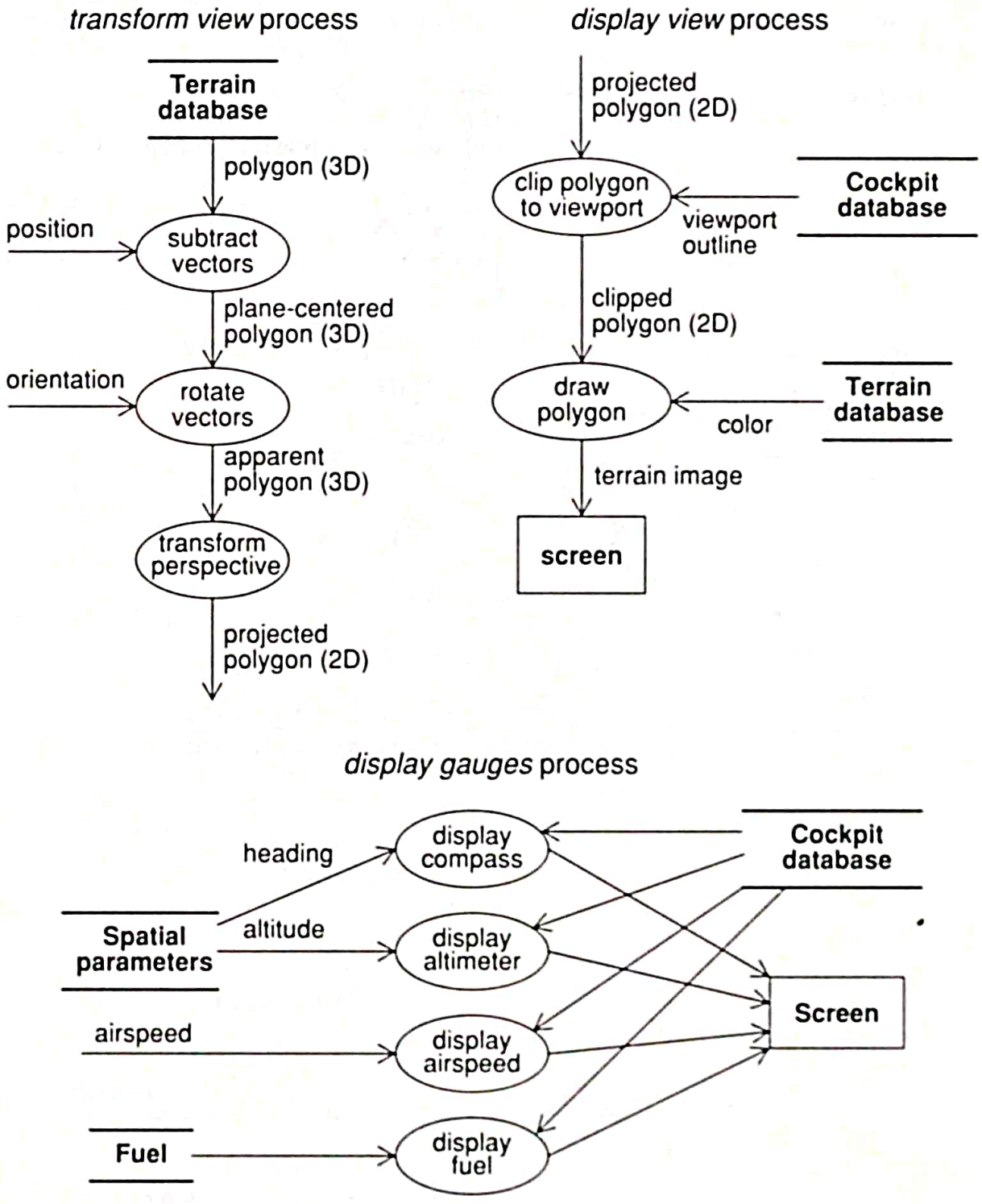

Figure 6.12 Expansion of display processes

from process *adjust controls*. The internal processes, such as *compute drag*, *compute lift*, and *compute density*, would be specified by aeronautical formulas and look-up tables for the specific airplane. For example, *compute lift* is specified by the equation $L = C(\alpha)S\rho V^2/2$, where L is lift, α is the angle of attack, S is the wing area, ρ is the air density, V is the airspeed, and C is the coefficient of lift as a function of angle of attack, specified by a table for the particular kind of wing. Process *integrate motion* is the solution to the differential equations of

motion. It is easy to specify, but its implementation involves careful numerical analysis considerations.

The display processes are *transform view*, *display view*, *display gauges*, and *display cockpit*. These processes convert the airplane parameters and terrain into a simulated view on the screen. They are expanded on Figure 6.12. Process *transform view* transforms the coordinates of a set of polygons in the *Terrain database* into the pilot's coordinate system, by first offsetting them by the plane's position, rotating the polygons by the plane's orientation, and then transforming the polygons' perspective onto the viewing plane to produce a 2-D image in the pilot's eye view. The position and orientation of the plane are input parameters. Process *display view* clips the image of the transformed polygons to remain within the output of the cockpit viewport, whose shape is specified in a *Cockpit database*. The clipped 2-D polygons are drawn on the screen using the colors specified in the *Terrain database*. Process *display gauges* displays various airplane parameters as gauges, using locations specified in the cockpit database. Process *display cockpit* displays a fixed image of the stationary parts of the cockpit and needed not be expanded.

Note that the functional model does not specify when, why, and how often values are computed. In a simulation such as this one, the motion integration might be performed more often than view computation because integration is subject to cumulative errors if too large an interval is used. Other computations can be omitted by sorting data cleverly. A smart view-mapping algorithm would quickly eliminate most of the terrain polygons using crude direction or distance checks so that only a few polygons would require costly full transformations and visibility checks. The set of active polygons would have to be updated occasionally as the plane moves, but hopefully not too often. Such considerations are part of the implementation algorithm but do not show up in the data flow diagram, which shows the underlying flow of data and computations but not the control decisions added by an implementation.

6.6 RELATION OF FUNCTIONAL TO OBJECT AND DYNAMIC MODELS

The functional model shows what "has to be done" by a system. The leaf processes are the operations on objects. The object model shows the "doers"—the objects. Each process is implemented by a method on some object. The dynamic model shows the sequences in which the operations are performed. Each sequence is implemented as a sequence, loop, or alternation of statements within some method. The three models come together in the implementation of methods. The functional model is a guide to the methods.

The processes in the functional model correspond to operations in the object model. Often there is a direct correspondence at each level of nesting. A top-level process corresponds to an operation on a complex object, and lower-level processes correspond to operations on more basic objects that are part of the complex object or that implement it. Sometimes one process corresponds to several operations, and sometimes one operation corresponds to several processes.

Processes in the functional model show objects that are related by function. Often one of the inputs to a process can be identified as the target object, with the rest being parameters

to the operation. The target object is a *client* of the other objects (called *suppliers)* because it uses them in performing the operation. The target knows about the clients, but the clients do not necessarily know about the target. The target object class is dependent on the argument classes for its operations. The client-supplier relationship establishes implementation dependencies among classes; the clients are implemented in terms of, and are therefore dependent on, the supplier classes.

A process is usually implemented as a method. If the same class of object is an input and an output, then the object is usually the target, and the other inputs are arguments. If the output of the process is a data store, the data store is the target. If an input of the process is a data store, the data store is the target. Frequently a process with an input from or output to a data store corresponds to two methods, one of them being an implicit selection or update of the data store. If an input or output is an actor, then it is the target. If an input is an object and an output is a part of the object or a neighbor of the object in the object model, then the object is the target. If an output object is created out of input parts, then the process represents a class method. If none of these rules apply, then the target is often implicit and is not one of the inputs or outputs. Often the target of a process is the target of the entire subdiagram. For example, in Figure 6.9, the target of *compute forces* is actually the airplane itself. Data stores *weight* and *spatial parameters* are simply components of the airplane accessed during the process.

Actors are explicit objects in the object model. Data flows to or from actors represent operations on or by the objects. The data flow values are the arguments or results of the operations. Because actors are "self-motivated" objects, the functional model is not sufficient to indicate when they act. The dynamic model for an actor object specifies when it acts.

Data stores are also objects in the object model, or at least fragments of objects, such as attributes. Each flow into a data store is an update operation. Each flow out of a data store is a query operation, with no side effects on the data store object. Data stores are passive objects that respond to queries and updates, so the dynamic model of the data store is irrelevant to its behavior. The dynamic model of the actors in a diagram is necessary to determine the order of operations.

Data flows are values in the object model. Many data flows are simply pure values, such as numbers, strings, or lists of pure values. Pure values can be modeled as classes and implemented as objects in most languages, but they do not have identity. A pure value is not a container whose value can change but just the value itself. A pure value therefore has no state and no dynamic model. Operations on pure values yield other pure values and have no side effects. Arithmetic operations are examples of such operations.

Other data flows represent normal objects. The selection data flow notation of Figure 6.5 explicitly produces a data store object to be operated on by other flows. Some input data flows to processes represent objects that are the targets of the processes. For example, the polygons in the top half of Figure 6.12 are the targets of several operations. In the object model, class *polygon* would have operations *subtract position*, *rotate*, *transform perspective*, and *clip to viewport*. In still other cases, a data flow represents an object that remains encapsulated; it is created by one process and passed through to another without change. Such data flows represent arguments to operations, rather than targets. For example, in Figure 6.6, the

input *account* to *bank* is an object that is stored within *bank* without being operated on; the same object is treated as a target object with respect to deposits from the customer.

Relative to the functional model: The object model shows the structure of the actors, data stores, and flows in the functional model. The dynamic model shows the sequence in which processes are performed.

Relative to the object model: The functional model shows the operations on the classes and the arguments of each operation. It therefore shows the supplier-client relationship among classes. The dynamic model shows the states of each object and the operations that are performed as it receives events and changes state.

Relative to the dynamic model: The functional model shows the definitions of the leaf actions and activities that are undefined with the dynamic model. The object model shows what changes state and undergoes operations.

6.7 CHAPTER SUMMARY

The functional model shows a computation and the functional derivation of the data values in it without indicating how, when, or why the values are computed. The dynamic model controls which operations are performed and the order in which they are applied. The object model defines the structure of values that the operations operate on. For batch-like computations, such as compilers or numerical computations, the functional model is the primary model, but in large systems all three models are important.

Data flow diagrams show the relationship between values in a computation. A data flow diagram is a graph of processes, data flows, data stores, and actors. Processes transform data values. Low-level processes are simple operations on single objects, but higher-level processes can contain internal data stores subject to side effects. A data flow diagram is a process. Data flows relate values on processes, data stores, and actors. Actors are independent objects that produce and consume values. Data stores are passive objects that break the flow of control by introducing delays between the creation and the use of data. As a rule, control information should be shown in the dynamic model and not the functional model, although control flows in data flow diagrams are occasionally useful.

Data flow diagrams can be nested hierarchically, but ultimately the leaf processes must be specified directly as operations. Operations can be specified by a variety of means, including mathematical equations, tables, and constraints between the inputs and outputs. An operation can be specified by pseudocode, but a specification does not imply a particular implementation; it may be implemented by a different algorithm that yields equivalent results. Operations have signatures that specify their external interface and transformations that specify their effects. Queries are operations without side effects; they can be implemented as pure functions. Actions are operations with side effects but without duration; they can be implemented as procedures. Activities are operations with side effects and duration; they must be implemented as tasks. Operations can be attached to classes within the object model and implemented as methods.