6

Advanced State Modeling

Conventional state diagrams are sufficient for describing simple systems but need additional power to handle large problems. You can more richly model complex systems by using nested state diagrams, nested states, signal generalization, and concurrency.

This is an advanced chapter and you can skip it upon a first reading of the book.

6.1 Nested State Diagrams

6.1.1 Problems with Flat State Diagrams

State diagrams have often been criticized because they allegedly are impractical for large problems. This problem is 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 diagrams, however, only $2n$ states are required.

Or consider the state diagram in Figure 6.1 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. Complex systems typically contain much redundancy that structuring mechanisms can simplify.

6.1.2 Expanding States

One way to organize a model is by having a high-level diagram with subdiagrams expanding certain states. This is like a macro substitution in a programming language. Figure 6.2 shows such a state diagram for a vending machine. Initially, the vending machine is idle. When a person inserts coins, the machine adds the amount to the cumulative balance. After adding some coins, a person can select an item. If the item is empty or the balance is insufficient, the machine waits for another selection. Otherwise, the machine dispenses the item and returns the appropriate change.

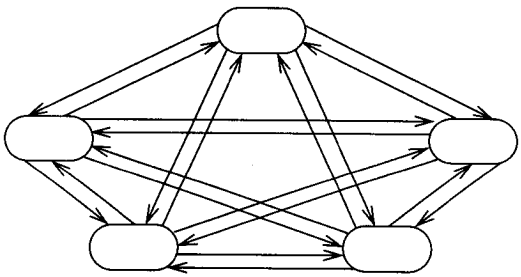

Figure 6.1 Combinatorial explosion of transitions in flat state diagrams. Flat state diagrams are impractical for large problems.

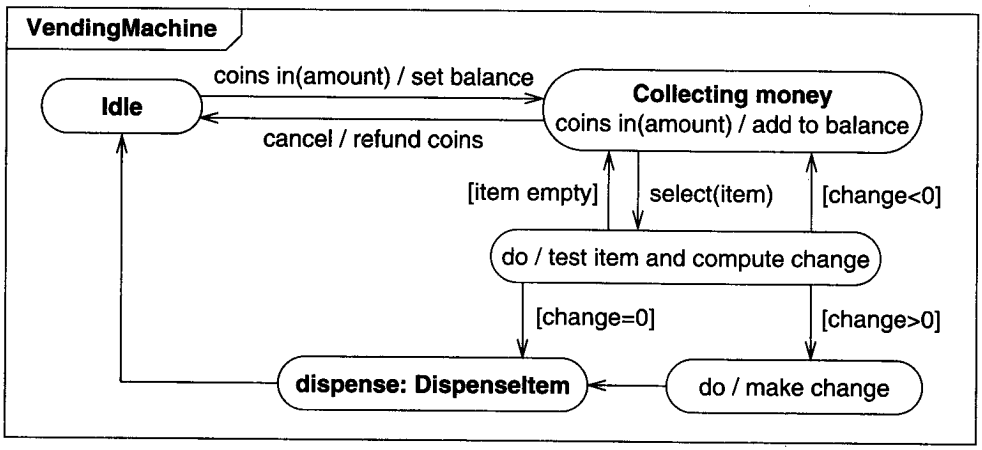

Figure 6.2 Vending machine state diagram. You can simplify state diagrams by using subdiagrams.

Figure 6.3 elaborates the *dispense* state with a lower-level state diagram called a submachine. A ***submachine*** is a state diagram that may be invoked as part of another state diagram. The UML notation for invoking a submachine is to list a local state name followed by a colon and the submachine name. Conceptually, the submachine state diagram replaces the local state. Effectively, a submachine is a state diagram "subroutine."

6.2 Nested States

You can structure states more deeply than just replacing a state with a submachine. As a deeper alternative, you can nest states to show their commonality and share behavior. (In accordance with UML2 we avoid using *generalization* in conjunction with states. See the *Bibliographic Notes* for an explanation.)

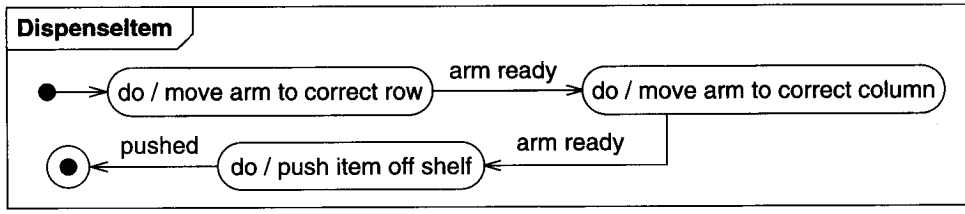

Figure 6.3 ***Dispense item*** **submachine of vending machine.** A lower-level state diagram can elaborate a state.

Figure 6.4 simplifies the phone line model from Chapter 5; a single transition from *Active* to *Idle* replaces the transitions from each state to *Idle.* All the original states except *Idle* are nested states of *Active.* The occurrence of event *onHook* in any nested state causes a transition to state *Idle.*

The ***composite state*** name labels the outer contour that entirely encloses the nested states. Thus *Active* is a composite state with regard to nested states *DialTone, Timeout, Dialing,* and so forth. You may nest states to an arbitrary depth. A nested state receives the outgoing transitions of its composite state. (By necessity, only ingoing transitions with a specified nested state can be shared, or there would be ambiguity.)

Figure 6.5 shows a state diagram for an automobile 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 nested states of state *Forward.*

Each of the nested states receives the outgoing transitions of its composite state. Selecting "N" in any forward gear shifts a transition to neutral. The transition from *Forward* to *Neutral* implies three transitions, one from each forward gear to neutral. Selecting "F" in neutral causes a transition to forward. Within state *Forward,* nested state *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.*

All three nested states share the transition on event *stop* from the *Forward* contour to state *First.* 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 nested state to a state outside the contour, or an explicit transition into the contour. In such cases, all the states must appear on one diagram. In simpler cases where there is no interaction except for initiation and termination, you can draw the nested states as separate diagrams and reference them by including a submachine, as in the vending machine example of Figure 6.2.

For simple problems you can implement nested states by degradation into "flat" state diagrams. Another option is to promote each state to a class, but then you must take special care to avoid loss of object identity. The *becomes* operation of Smalltalk lets an object change class without a loss of identity, facilitating promotion of a state to a class. However, the performance overhead of the *becomes* operation may become an issue with many state changes. Promotion of a state to a class is impractical with C++, unless you use advanced techniques, such as those discussed in [Coplien-92]. Java is similar to C++ in this regard.

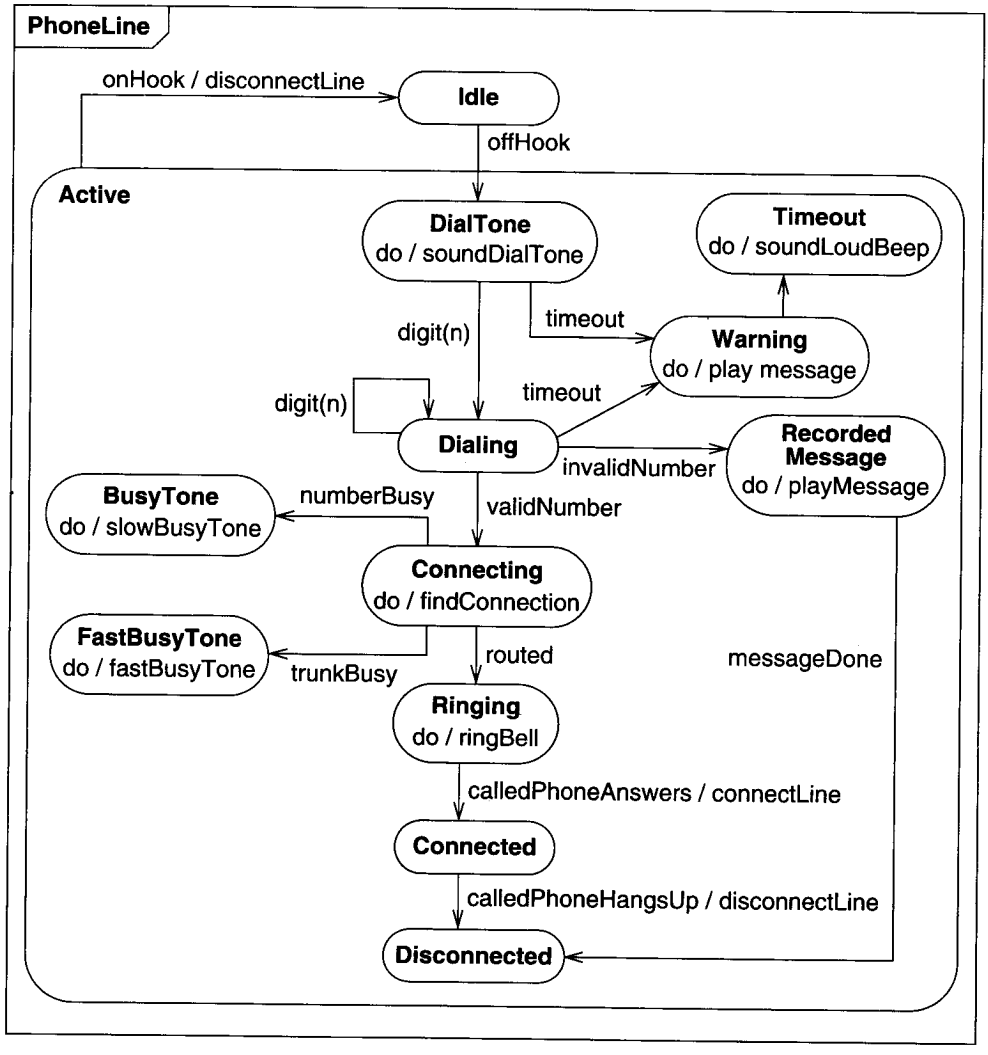

Figure 6.4 Nested states for a phone line. A nested state receives the outgoing transitions of its enclosing state.

Entry and exit activities 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 activities without regard for what happens before or after the state is active. Transitioning into or out of a nested state can cause execution of several entry or exit activities, if the transition reaches across several levels of nesting. The entry activities are executed from the outside in and the exit activities from the inside out. This permits behavior similar to nested subroutine calls.

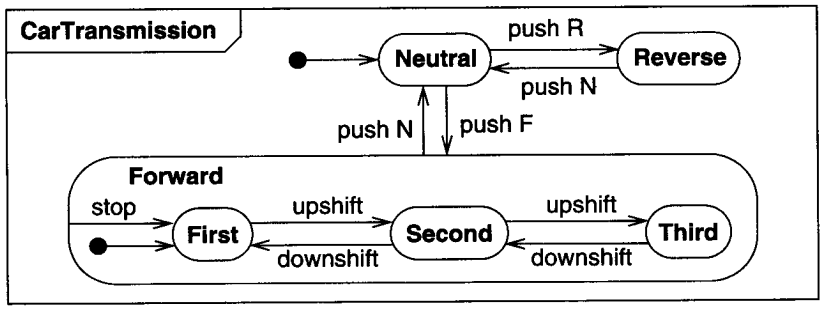

Figure 6.5 Nested states. You can nest states to an arbitrary depth.

6.3 Signal Generalization

You can organize signals into a generalization hierarchy with inheritance of signal attributes. Figure 6.6 shows part of a tree of input signals for a workstation. Signals *MouseButton* and *KeyboardCharacter* are two kinds of user input. Both signals inherit attribute *device* from signal *UserInput* (the root of the hierarchy). *MouseButtonDown* and *MouseButtonUp* inherit *location* from *MouseButton. KeyboardCharacters* can be divided into *Control* and *Graphic* characters. Ultimately you can view every actual signal as a leaf on a generalization tree of signals. In a state diagram, a received signal triggers transitions that are defined for any ancestor signal type. For example, typing an 'a' would trigger a transition on signal *Alphanumeric* as well as signal *KeyboardCharacter.* Analogous to generalization of classes, we recommend that all supersignals be abstract.

A signal hierarchy permits different levels of abstraction to be used in a model. For example, some states might handle all input characters the same; other states might treat control characters differently from printing characters; still others might have different activities on individual characters.

6.4 Concurrency

The state model implicitly supports concurrency among objects. In general, objects are autonomous entities that can act and change state independent of one another. However, objects need not be completely independent and may be subject to shared constraints that cause some correspondence among their state changes.

6.4.1 Aggregation Concurrency

A state diagram for an assembly is a collection of state diagrams, one for each part. The aggregate state corresponds to the combined states of all the parts. 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 part

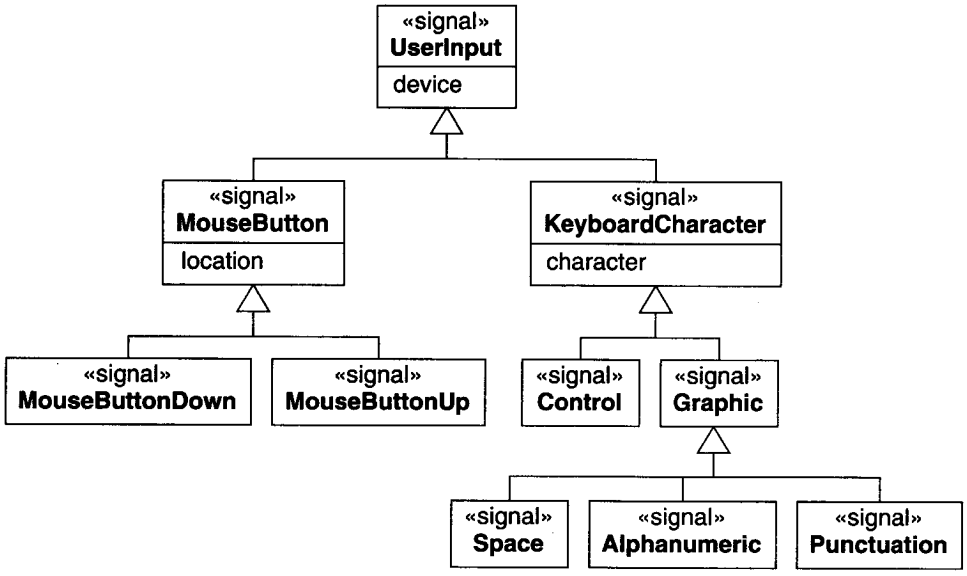

Figure 6.6 Partial hierarchy for keyboard signals. You can organize signals using generalization.

states interact. 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 6.7 shows the state of a *Car* as an aggregation of part states: *Ignition*, *Transmission*, *Accelerator*, and *Brake* (plus other unmentioned objects). The state of the car includes one state from each part. Each part undergoes transitions in parallel with all the others. The state diagrams of the parts are almost, but not quite, independent—the car will not start unless the transmission is in neutral. This is shown by the guard expression *Transmission in Neutral* on the transition from *Ignition-Off* to *Ignition-Starting*.

6.4.2 Concurrency within an Object

You can partition some objects 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. The UML shows concurrency within an object by partitioning the composite state into regions with dotted lines. You should place the name of the composite state in a separate tab so that it does not become confused with the concurrent regions.

Figure 6.8 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 regions are initially in their respective default states *Not vulnerable*. Each region can independently advance to state *Vulnerable*

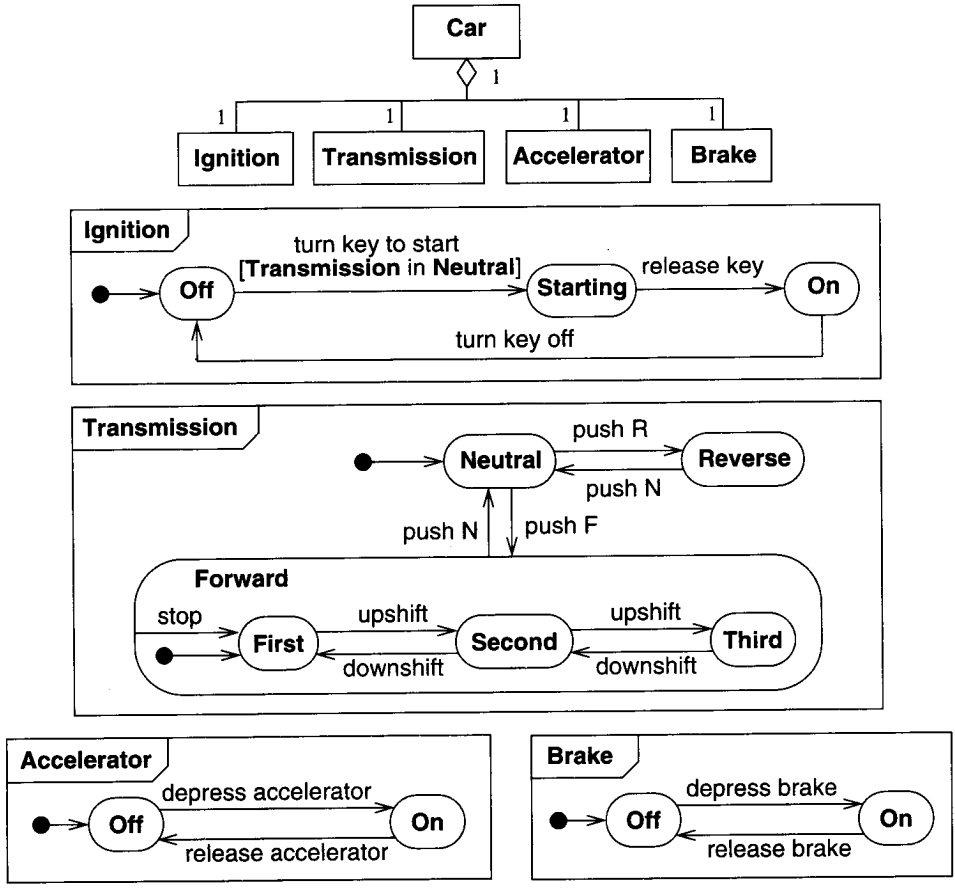

Figure 6.7 An aggregation and its concurrent state diagrams. The state diagram for an assembly is a collection of state diagrams, one for each part.

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 regions, because they are part of the same composite state *Playing rubber* and are active only when the top-level state diagram is in that state.

Most programming languages lack intrinsic support for concurrency. You can use a library, operating system primitives, or a DBMS to provide concurrency. During analysis you should regard all objects as concurrent. During design you devise the best accommodation; many implementations do not require concurrency, and a single thread of control suffices.

6.4.3 Synchronization of Concurrent Activities

Sometimes one object must perform two (or more) activities concurrently. The object does not synchronize the internal steps of the activities but must complete both activities before it can

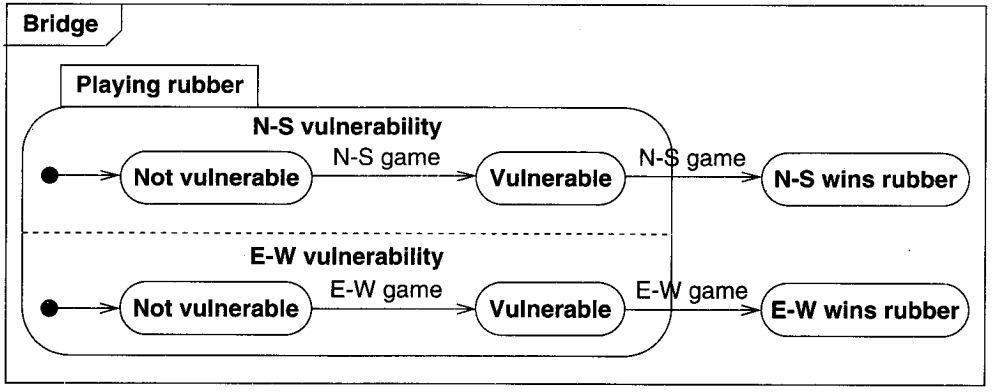

Figure 6.8 Bridge game with concurrent states. You can partition some objects into subsets of attributes or links, each of which has its own subdiagram.

progress to its next state. For example, a cash dispensing machine 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 6.9 shows a concurrent state diagram for the emitting activity. The number of concurrently active states varies during execution from one to two and back to one again. The UML shows concurrent activities within a single composite activity 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 consists of exactly one state from each subdiagram.

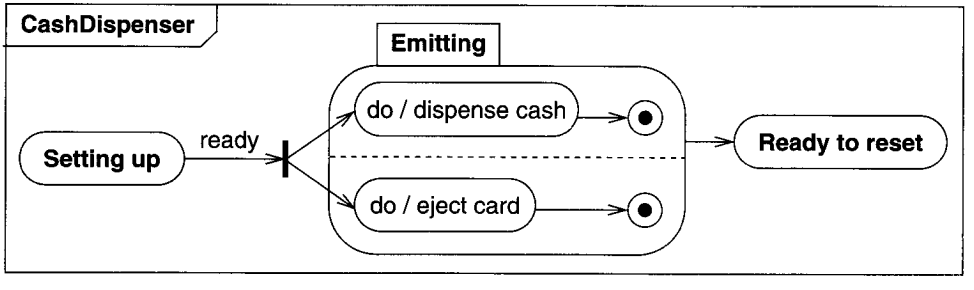

Figure 6.9 Synchronization of control. Control can split into concurrent activities that subsequently merge.

A transition that forks indicates splitting of control into concurrent parts. A small heavy bar with one input arrow and two or more output arrows denotes the fork. The event and an optional guard condition label the input arrow. The output arrows have no labels. Each output

arrow selects a state from a different 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.

Any transition into a state with concurrent subdiagrams activates each of the subdiagrams. If the transition omits any subdiagrams, the subdiagrams start in their default initial states. In this example, a forked arrow is not actually necessary. You could draw a transition to the *Emitting* state, with each subdiagram having a default initial state.

The UML shows explicit merging of concurrent control by a transition with two or more input arrows and one output arrow, all connected to a small heavy bar (not shown in Figure 6.9). The trigger event and optional guard condition are placed near the bar. The target state becomes active when all of the source states are active and the trigger event occurs. Note that the transition involves a single event, not one event per input arrow. If any subdiagrams in the composite state are not part of the merge, they automatically terminate when the merge transition fires. As a consequence, a transition from a single concurrent substate to a state outside the composite state causes the other concurrent substates to terminate. You can regard this as a degenerate merge involving a single state.

An unlabeled (completion) transition from the outer composite state to another state indicates implicit merging of concurrent control (Figure 6.9). A completion transition fires when activity in the source state is complete. A composite concurrent state is complete when each of its concurrent substates is complete—that is, when each of them has reached its final state. All substates must complete before the completion transition fires and the composite state terminates. In the example, when both activities have been performed, both substates are in their final states, the merge transition fires, and 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. The firing of a merge transition causes a state diagram to perform the exit activities (if any) of all subdiagrams, in the case of both explicit and implicit merges.

6.5 A Sample State Model

We present a sample state model of a real device (a Sears "Weekender" Programmable Thermostat) to show how the various modeling constructs fit together. We constructed this model by reading the instruction manual and experimenting with the actual device. The device controls a furnace and air conditioner according to time-dependent attributes that 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 values at the beginning of each program period. The table specifies the target temperature and start time for eight different time periods, four on weekdays and four on weekends. The user can override the target temperature.

The user programs the thermostat using a pad of ten pushbuttons and three switches and sees parameters on an alphanumeric display. Each pushbutton 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 eight 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:

NIGHT LIGHT	Lights the alphanumeric display. Values: light off, light on.
SEASON	Specifies which device the thermostat controls. Values: heat (furnace), cool (air conditioner), off (none).
FAN	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 sensor for air temperature 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 state model, it is important to include only states that affect the flow of control and to model other information as parameters or variables. We introduce an internal state variable *target temp* to represent the current temperature that the thermostat is trying to maintain. Some activities read this state variable and others set it; the state variable permits communication among parts of the state model.

Figure 6.10 shows the top-level state diagram of the programmable thermostat. It contains seven concurrent subdiagrams. The user interface is too large to show and is expanded separately (Figure 6.11). The diagram includes trivial subdiagrams for the season switch and the fan switch. The other four subdiagrams show the output of the thermostat: the furnace, air conditioner, the run indicator light, and fan relays. Each of these subdiagrams contains an *Off* and an *On* substate. The state of each subdiagram is totally determined by input parameters and the state of other subdiagrams, such as the season switch or the fan switch. The state of the four subdiagrams on the right is totally derived and contains no additional information.

Figure 6.11 shows the subdiagram for the user interface. The diagram contains three 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 pushbutton 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. State *Operate* has three concurrent substates—one includes *Run* and *Hold*, another controls the target temperature display, and the third controls the time and temperature display. Every two seconds the display alternates between the current time and current temperature. 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.

After 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 activity 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. If the interface is in one of the setup states for 90 seconds without any input, the system enters the *Hold* state. 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 nested states for the 90-second timeout 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 6.12. Pressing *set clock* enters the *Set minutes* nested state as initial default. Subsequent *set clock* presses toggle between the *Set hours* and the *Set minutes* nested states. The *time fwd* and *time back* buttons modify the program time. Pressing *set day* enters the *Set day* nested state and shows the day of the week. Subsequent presses increment the day directly.

Pressing *view program* enters the *Set program* nested state, 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* nested states has an explicit exit transition. Each nested state is implicitly terminated by a transition into another nested state from the main *Interactive display* contour.

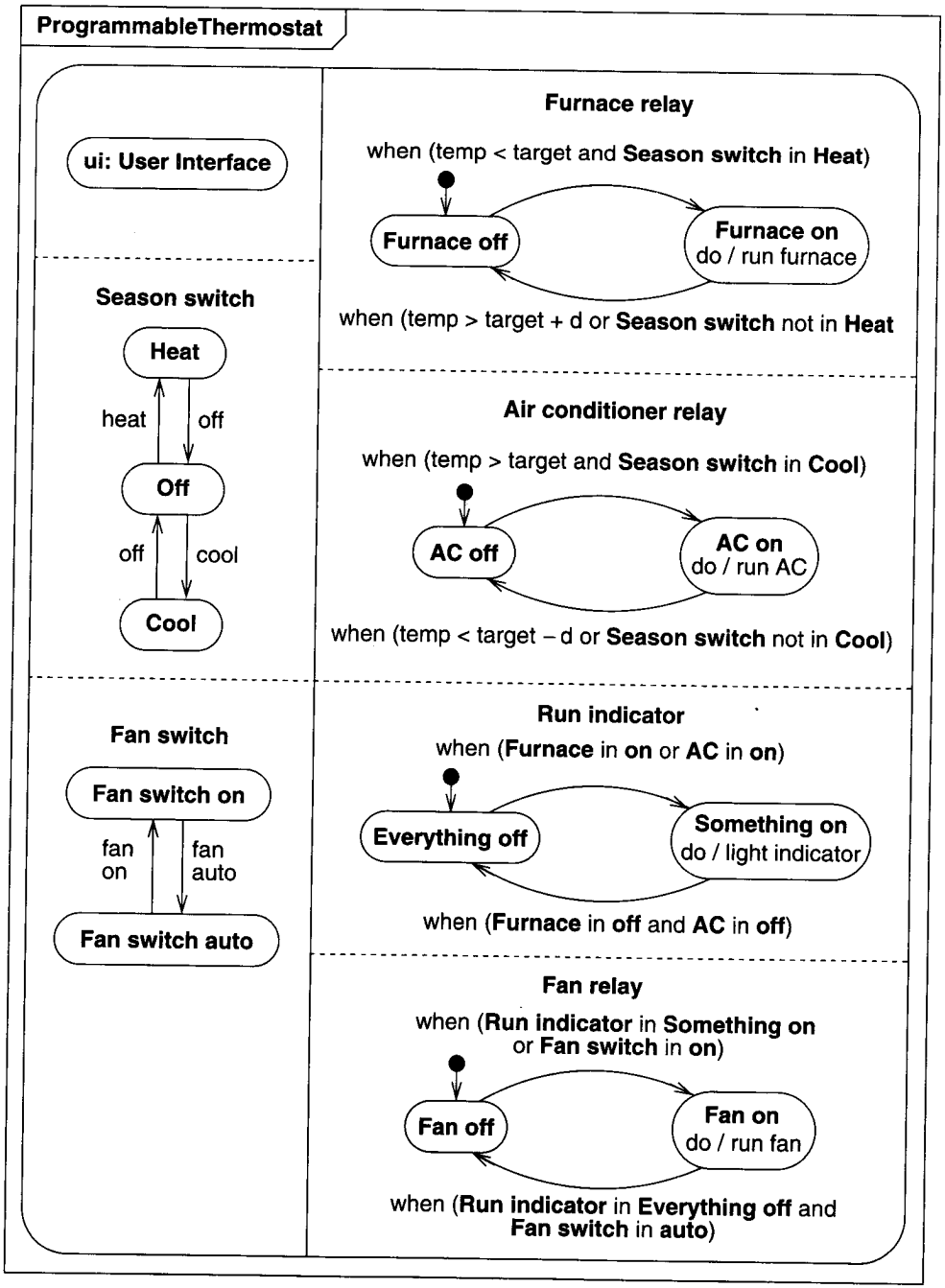

Figure 6.10 State diagram for programmable thermostat

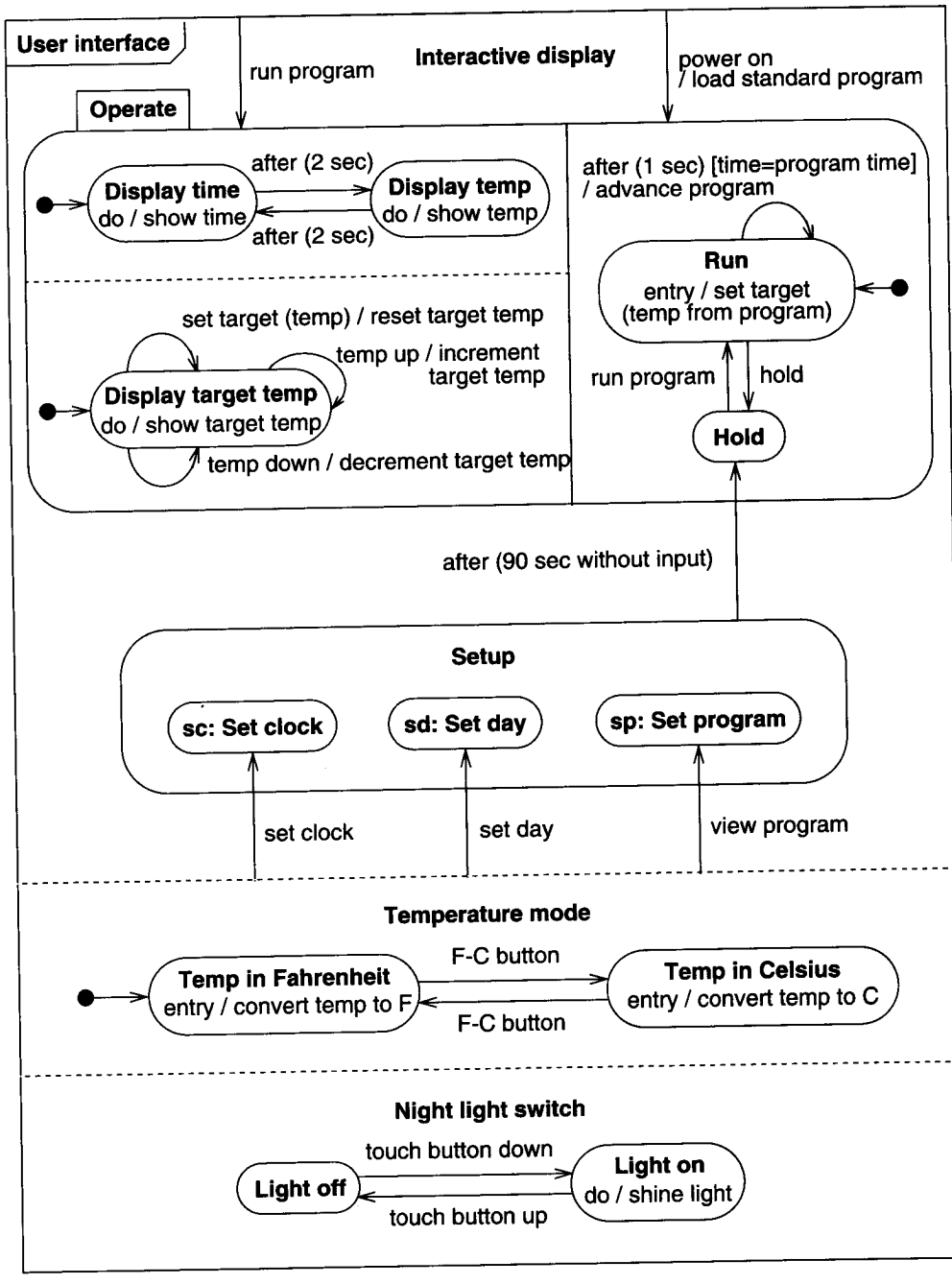

Figure 6.11 Subdiagram for thermostat user interface

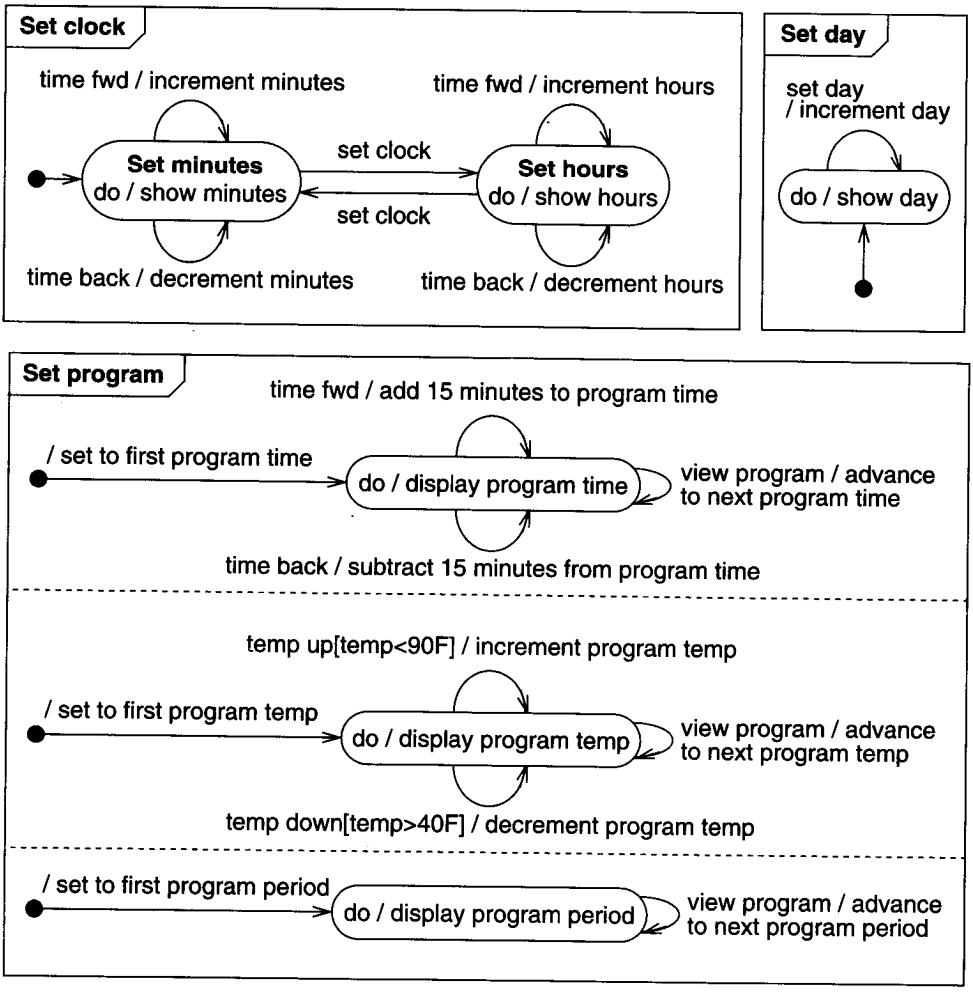

Figure 6.12 Subdiagrams for thermostat user interface setup

6.6 Relation of Class and State Models

The state model specifies allowable sequences of changes to objects from the class model. A state diagram describes all or part of the behavior of the objects of a given class. States are equivalence classes of values and links for an object.

State structure is related to and constrained by class structure. A nested state refines the values and links that an object can have. Both generalization of classes and nesting 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 differ-

ences 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 class model. The first is aggregation of objects: Each part of an aggregation has its own independent state, so the assembly can be considered to have a state that is the combination of the states of all its parts. The second source is aggregation within an object: The values 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 6.9. 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 state 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? 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 elaborated with nesting 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 signal hierarchy is independent of the class hierarchy for the classes exchanging signals, in practice if not in theory. Signals can be defined across different classes of objects. Signals 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, with the operation name corresponding to the signal name. Signals are more expressive than operations, however, because the effect of a signal depends not only on the class of an object but also on its state.

6.7 Practical Tips

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

- **Structured state diagrams.** Use structure to organize models with more than 10–15 states. (Section 6.1)
- **State nesting.** Use nesting when the same transition applies to many states. (Section 6.2)

- **Concrete supersignals**. Analogous to generalization of classes, it is best to avoid concrete supersignals. Then, abstract and concrete signals are readily apparent at a glance—all supersignals are abstract and all leaf subsignals are concrete. You can always eliminate concrete supersignals by introducing an *Other* subsignal. (Section 6.3)
- **Concurrency**. Most concurrency arises from object aggregation and need not be shown explicitly in the state diagram. Use composite states to show independent facets of the behavior of a single object. (Section 6.4)
- **Consistency of diagrams**. Check the various state diagrams for consistency on shared events so that the full state model will be correct. (Section 6.5)
- **State modeling and class inheritance**. Try to make the state diagrams of subclasses independent of the state diagrams of their superclasses. Subclass state diagrams should concentrate on attributes unique to the subclasses. (Section 6.6)

6.8 Chapter Summary

A class model describes the objects, values, and links that can exist in a system. The 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. Objects respond to events, which are occurrences at a point in time. 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 a signal to the original sender or to a third object.

The combinations 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 a class diagram is a network of classes and relationships. The state model consists of multiple state diagrams, one state diagram for each class with important dynamic behavior, and shows the possible behavior for an entire system. Each object independently executes the state diagram for its class. The state diagrams for the various classes communicate via shared events.

States and events can both be expanded to show greater detail. Nested states share the transitions of their composite states. Signals can be organized into inheritance hierarchies. Subsignals trigger the same transitions as their supersignals.

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

Entry and exit activities permit activities to cover all the transitions entering or exiting the state. They make self-contained state diagrams possible for use in multiple contexts. Internal activities represent transitions that do not leave the state.

A subclass inherits the state diagrams of its ancestors, to be concurrent with any state diagram that it defines. It is also possible to refine an inherited state diagram by expanding states into nested states or concurrent subdiagrams.