2667-ch07 10/16/02 8:44 AM Page 75

CHAPTER

7 Describing Behaviors Using Statecharts

OBJECTIVES

In this chapter you will:

- Learn to use the adapted statechart notation
- Explore using context hierarchy and concurrency in the design of simulators
- Learn to construct event-action tables
- Discover how statecharts and event-action tables map to ActionScript

Constructing a simulation without a complete, well thought out design is like building a house on a weak foundation. This chapter introduces the sound foundation: a visual design methodology and notation, called statecharts, for managing the complexities of designing simulators, such as medical, aviation, and consumer electronics devices. It was developed by David Harel in the 1980s to assist in building airplane simulators for the Israeli aircraft industry. It is an extension of deterministic finite state automata (or state machines) from the field of computer science. More recently, statecharts have been adopted into the Unified Modeling Language (UML).

The theoretical information in this chapter may not seem so readily applicable until you sit down and start to design your own simulation. Therefore, we recommend that you revisit this chapter when you either have difficulty understanding its application in other parts of the book or need a reference as far as notation or coding is concerned.

Statecharts visually convey how a device can behave. Among their strengths are the following two critical aspects.

- Statecharts permit rapid and robust coding directly from the visual description and accompanying event-action tables. This chapter and the examples in later chapters will show you how to do this in Flash MX.
- The statechart and tables provide excellent documentation and validation tools that ensure that if you need to extend or alter the simulator you know immediately where it should be done and how to do it.

Furthermore, the notation is easy to learn and enables subject matter experts who do not know programming to design the complete simulator and hand off the design to programmers who implement it in a cookie-cutter fashion. Coupling this methodology with Flash MX components (described in Part III) can dramatically reduce implementation time.

In this chapter, you are introduced to the key concepts of statecharts and the appropriate notation for each concept. Examples illustrate the subtleties and richness of the notation, and code examples demonstrate the ActionScript implementation used in the rest of this book.

NOTE: The Flash MX implementation is the cornerstone of the examples and projects in this book. It is located on the companion CD-ROM in the *FStEng* folder, in a file called *FStEng.as*. The file *TestCases.fla*, in the same location, tests the state engine in a variety of complex configurations. Appendices B through E provide documentation on the state engine implementation. The companion CD-ROM also contains a template for drawing statecharts using the SmartDraw program on the PC (demo included on the companion CD-ROM). The template is located in the *Statecharts for Smartdraw* folder.

WHAT IS A STATECHART?

A statechart is a complete graphical characterization of a system's potential behavior to the level of detail required for the simulation. It consists of discrete "states" and "transitions." Each state represents a distinct context for behaviors of the device, such as an ON state and an OFF state. As discussed in Chapter 6, the event handler uses the current context to determine how to interpret incoming events. For example, turning a knob has a different meaning when the device is on, compared with when the device is off. In addition to its use in determining the context, each state can have actions associated with it, such as performing the function of the device when the device is in that context. Therefore, the device's behavior is determined by which context it is in at the moment.

States may be hierarchical in that a single context at one level, such as the ON state, can have distinct subcontexts represented as substates. These immediate substates form a state network. Like a good road map, a statechart provides just enough detail to meet the intended use. For example, a high-level statechart might include only one or two levels of detail, providing a bird's-eye view of the overall system's behavior.A *transition* is a directed connection between any two states that dictates under what conditions (the "trigger") the device shifts from the first context (state) to the second.

Statecharts can also serve as powerful communication tools when used by members of large teams to discuss the relationships of subsystems within a larger supersystem. An airplane, for example, can be conceptualized as a single complex device. A statechart to describe an airplane as a whole at a useful level of detail would, however, be unnecessarily complex. Instead, the airplane can be treated as a set of integrated subsystems. One statechart can be used to describe the pneumatics system, and another the hydraulics system, until all subsystems can be combined into an integrated whole. As an introduction to statecharts, the text weaves together the pieces into a complete notation, and provides ActionScript coding examples.

2667-ch07 10/16/02 8:44 AM Page 77

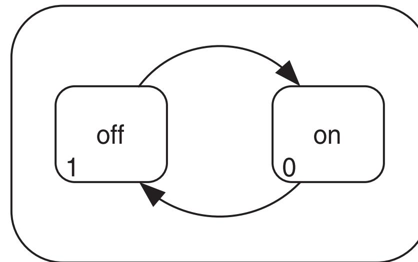


FIGURE 7–1
Macro-level states of a flashlight. As a coding convention, we put the state ID (0 and 1, in this case) in the lower left corner, although this is not part of Harel's notation.

STATES

States represent different contexts in which device behaviors occur. On a statechart, states are denoted by a rounded square symbol, as shown in figure 7–1. States have status, meaning that they can exist in an active or inactive state. When a state is active, the device is said to be "in that state." States also serve as containers for code to implement behavior in that context. The system state of the entire device at any moment (a snapshot) consists of the set of active states within all state networks.

The Macro Level

Let's first consider the behavior of an ordinary household flashlight. At a macro (high) level, a flashlight possesses two fundamentally different states (ON and OFF), as shown in figure 7–1. When the flashlight is in the OFF state, the lamp is extinguished. In the ON state, the lamp illuminates. The ON and OFF states of a simple flashlight are different and therefore warrant a state network with two distinct states.

The Micro Level

Simulated devices often possess micro-level behaviors, which also need to be described. At some level, small changes in behavior should not be represented by different states. For instance, consider a fancy flashlight equipped with a dimmer control. Although the dimmer control certainly allows the user to control the device, the flashlight is more compactly described with two states, ON and OFF, and the micro-changes represented by a quantitative value in the ON state.

Unlike finite state automata, states in statecharts can be hierarchical, meaning that within a state (such as ON) there can be several substates, each representing subcontexts for behaviors. For example, a device when turned on may

2667-ch07 10/16/02 8:44 AM Page 78

have a ramp-up period before achieving full power, and may have a ramp-down period before turning off. If the functionality available is different during these three periods (ramp-up, full, and ramp-down), you could model them as three substates of ON. Hierarchical states are important for grouping related actions that occur regardless of the active substate.

Coding Simple States

We call nonhierarchical states "simple" states. To create the states shown in figure 7–1, you would issue the following.

```
s0 = new state("on", 0, <sm>);
s1 = new state("off", 1, <sm>);
```

The *state* constructor builds an instance of a simple state, with the string name for the state (for debugging) being the first argument, the *state* identifier being the second argument, and the third being a pointer to the state (network) manager. It is important to hold onto the state instance because you will use that variable to add transitions and other state code. By convention, the *state* identifier is a positive integer that is different from other IDs of sibling states (i.e., states at the same level). In this example, <*sm*> is a placeholder for a pointer you put to the state's network manager (discussed in material to follow). The manager coordinates which state is currently active within a state network.

Hierarchical States

One of the powerful features of statecharts is that they allow designers to group related contexts as hierarchical states. Hierarchical states show the relatedness of contexts. For example, it is acceptable in the flashlight example to draw a hierarchical state labeled *power available* around the two-state flashlight statechart, as is the case when the flashlight has fresh batteries. You can further add a simple state, labeled no *power available,* to describe the device without batteries. By configuring the statechart in this manner, you can describe the fact that the ON state is a function of both the switch being moved to the on position and the system having power available. This scenario is shown in figure 7–2.

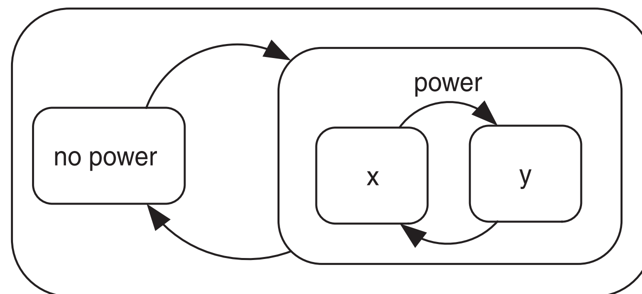


FIGURE 7–2

A hierarchical state *power,* with a state network consisting of two states. For clarity, we have omitted the state IDs.

2667-ch07 10/16/02 8:44 AM Page 79

The hierarchical state that contains other states is called the *parent* state of the contained children (sub-) states. A child state cannot be active if the parent is inactive. As parent states change from an inactive state to an active state, at least one child state necessarily becomes active.

Coding Hierarchical States

A hierarchical state is created in a manner similar to that of a simple state, as in the following.

```
powState = new hstate("power", 5, <sm>);
```

Here again, <*sm*> is a pointer to the state's network manager. The state ID has to be unique for sibling states, but can be the same as IDs for any substate, sub-substate, and so on. The major difference between coding a simple and hierarchical state is that when you code a hierarchical state you must also define a state manager to keep track of the state network's run-time status. The basic syntax follows.

```
powerMgr = new state_mgr("A", "power mgr", 1, powState);
```

This syntax is explained in the section on state network managers in material to follow.

NOTE: The state manager denoted by <*sm*> in the previous code is not *powerMgr*. The former state manager is the manager of the network in which *powState* participates, whereas *powerMgr* is the manager of *powState's* substates.

The Root State

Every statechart begins with a root state that encloses the entire state network. Some designers will elect not to actually draw the root state on their statechart, but it does exist and needs to be coded. The root state is conceptually always active when the simulation is active.

Some large or especially complex devices may require you to model subsystems separately. After the subsystems have been designed, you can join the subsystems by creating an outermost root state around the subsystem statecharts. In figures 7-1 and 7-2, the root (outermost) states, which are hierarchical, could each be defined as follows.

```
stateNet = new hstate("root", 0, null);
```

The third argument indicates that the root state has no state manager, in that it does not participate in a state network. Note that because this example is a hierarchical state (typical for root), we use the *hstate* constructor. Therefore, it will also require a state network manager.

2667-ch07 10/16/02 8:44 AM Page 80

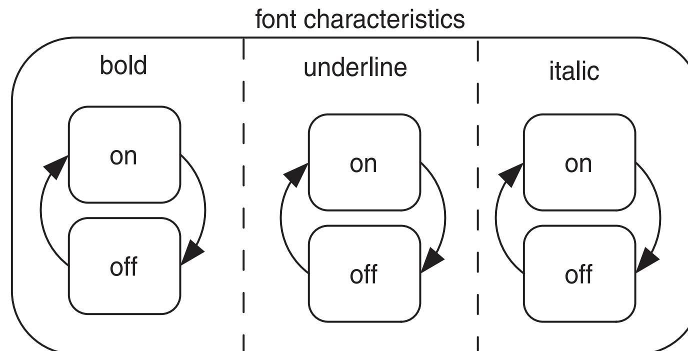


FIGURE 7–3
A statechart representing the states of boldface, underscore, and italic buttons.

Simple versus Concurrent Hierarchical States

Hierarchical states can be what we call simple (exclusive) or concurrent. A simple hierarchical state means that the state's network consists of substate contexts that are mutually exclusive. This means that the device can be in only one of these substates at any moment. The hierarchical state only needs one network manager to keep track of which state is active. Both the "power" state and the root state are hierarchical states, as shown in figure 7–2.

Alternatively, a hierarchical state can have concurrent networks. A concurrent hierarchical state means that the state contains two or more networks, each of which has a set of mutually exclusive states and is separated from others by a dashed line. These states enable you to model related but independent parts of systems.

For example, in modeling a word processor interface, you could use the statechart shown in figure 7–3 to describe font characteristic buttons such as boldface, underscore, and italics, each of which can be in the on or off position.

The buttons are related to one another because they represent font characteristics; hence, the resulting text will be influenced by which state is active within each network. However, whether the boldface button is in the on or off position is completely independent of the position of the italic or underscore button.

Chapter 15 describes how to construct a robotic arm. A more advanced arm could be designed to have three independent movement characteristics: arm rotation, arm extension, and claw clasp. Because these actions can occur independently of one another, we would model them in concurrent states. In this design, the arm could rotate at the same time it extends. One movement is entirely independent of the other two.

Coding Simple and Hierarchical States

You have already seen how to code a simple hierarchical state. To code a concurrent hierarchical state, you use the *hstate_c* class, as follows.

```
fontCharButs = new hstate_c("font characteristic buttons", 0, <sm>);
```

2667-ch07 10/16/02 8:44 AM Page 81

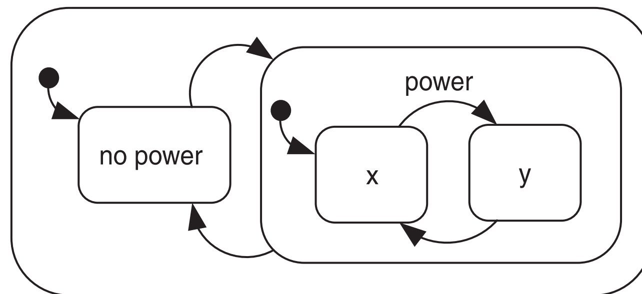


FIGURE 7–4
The flashlight statechart with default start states marked.

You then add one state manager for each substate network. Following that, you define each substate as belonging to one of the managers; for example, as follows.

```
boldOn = new state("bold on", 0, boldMgr);
undOn = new state("underline on", 5, undMgr);
```

NOTE: All immediate substates of *fontCharButs* must have unique state IDs, even if they belong to different state networks.

Default Start State

Each state network must designate a single state as the state in which to activate when the network is activated (i.e., the hierarchical state that contains the network is activated). This is known as the default start state, and the notation is a stubbed arrow pointing to the default start state, as shown in figure 7–4.

The default start state represents the first context that should be activated when the hierarchical state is activated. In concurrent hierarchical states, each state network requires a default state.

State Network Managers

In a state network at runtime, a state (simple or hierarchical) can be either active or inactive. The statechart shows all possible states for the device, but it does not show how the network at runtime keeps track of which states are active and which are inactive. This is the role of the state network manager, or more simply, the state manager. The state manager is not part of Harel's statechart notation, because the notation does not describe runtime behavior of the network, but we have added it because it is necessary for the implementation.

A state manager is responsible for a state network. It keeps track of which state is current (if any), which state should be entered by default when the hierarchical state is entered, and which state was most recently visited. Hierarchical states have one or more state networks, but simple states have none. Therefore, only hierarchical states need state managers.

A state manager is usually denoted on a statechart by a single letter in the upper right-hand corner of the state to distinguish it from a state ID, as in

2667-ch07 10/16/02 8:44 AM Page 82

FIGURE 7–5

The flashlight statechart with state manager IDs and state IDs marked.

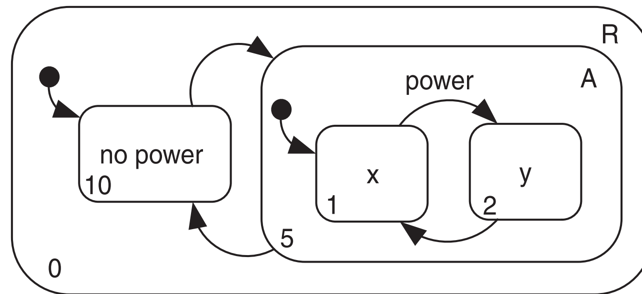


figure 7–5. It is not required that state managers at different network levels have different IDs, but it may make understanding the code resulting from the statechart easier.

Coding State Managers

Let’s revisit the powerMgr state manager definition, which follows.

```
powerMgr = new state_mgr(“A”, “power mgr”, 1, powState);
```

In the code that follows, the root state defines state manager *R,* which has its default start state set to the *no power* (ID 10) state. The *power* state, ID = 5, defines state manager *A.* The state manager takes four arguments. The first argument is the manager ID, which by convention is a letter. The second argument is a string to use for debugging purposes when the manager is activated or deactivated. The third argument gives the state ID of the default start state. The last argument gives the hierarchical state to which the manager belongs.

NOTE: The order of ID and debugging name, the first two parameters, is the reversal of these parameters for state definitions.

The complete code for the states and managers shown in figure 7–5 is as follows.

```
stateNet = new hstate(“root”, 0, null);
rootMgr = new state_mgr(“R”, “root mgr”, 10, stateNet);
noPowState = new state(“no power”, 10, stateNet);
powState = new hstate(“power”, 5, stateNet);
powerMgr = new state_mgr(“A”, “power mgr”, 1, powState);
stX = new state(“x”, 1, powerMgr);
stY = new state(“y”, 2, powerMgr);
```

NOTE: Because the manager requires knowing which network it is to manage, you must define the manager after you define the hierarchical state.

Concurrent hierarchical states are a little more complex because they contain two or more state networks, each of which requires state managers. Thankfully, the code is the same regardless of the hierarchical state type. The constructor

2667-ch07 10/16/02 8:44 AM Page 83

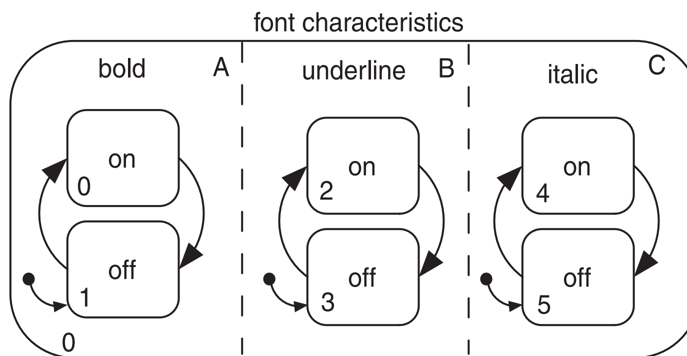


FIGURE 7–6
The word processing interface with state manager IDs and state IDs marked.

sorts out whether it is being attached to a simple or concurrent hierarchical state. However, you need to ensure that a state belongs only to a single network. Revisiting the word processing interface, figure 7–6 shows the network indicating the default states and all IDs of all managers and states.

The code for the concurrent hierarchical state is as follows (assuming the state is managed by an unseen *fcbMgr* state manager).

```
fontCharButs = new hstate_c("font characteristic buttons", 0, fcbMgr);
boldMgr = new state_mgr("A", "boldface mgr", 1, fontCharButs);
boldOn = new state("bold on", 0, boldMgr);
boldOff = new state("bold off", 1, boldMgr);
undMgr = new state_mgr("B", "underline mgr", 3, fontCharButs);
undOn = new state("underline on", 2, undMgr);
undOff = new state("underline off", 3, undMgr);
itMgr = new state_mgr("C", "italics mgr", 5, fontCharButs);
itOn = new state("italics on", 4, itMgr);
itOff = new state("italics off", 5, itMgr);
```

Advanced Peek into States and State Managers

Appendix D (on the companion CD-ROM) summarizes the methods and properties of the states and managers. Naturally, hierarchical states are subclasses of states. A hierarchical state has a *substates* property that holds all substates. A simple hierarchical state has a property (named *st_mgr*) to hold the state manager for its network. A concurrent hierarchical state has a property (called *st_mgrs*) to hold a list of its networks' state managers. Each state manager has a property (called *cs*) that holds the ID of the currently active state, and a property (called *def_st*) that holds the default start state ID (remember, the actual states are in the *substates* property of the hierarchical state).

The implication of this is that you can either hold onto the states you create as you create them, or you can use this data structure to find the state or manager object when you need it. If you want more detail than given in Appendix D, consult the source code located in the *FStEng* folder of the companion CD-ROM.

TRANSITIONS

Having established the different contexts for device behavior, you now need to specify when and to which contexts the device can switch. This is the job of transitions. Transitions represent potential pathways among states of devices. Transitions are "fired" when they are "triggered," meaning that some event or condition (the trigger) is satisfied. The event handlers are typically responsible for testing the triggers and firing the appropriate state transitions for the current context.

Transitions are represented on statecharts as a line with an arrow drawn from the source state to the target state. In a high-level sketch of a statechart, you can place the trigger text directly on the transition.

Simple Transition

A simple transition is a transition from a state to a sibling state, as shown in figure 7–7. In other words, both source state and target state exist at the same level of the state network.

You code simple transitions by adding them to the source state, giving them unique IDs with respect to other transitions from that state, as follows.

```
s1.add_trans(2, new trans(null, 0));
```

Following the example in figure 7–7, this code adds a simple transition with ID = 2 to state 1. The first parameter is the transition ID. The second parameter is a simple transition object that has two parameters. The first parameter is a function to be invoked when the transition is fired (discussed in material to follow). The second parameter is the ID of the target state, which in this case is 0. It is very important to recognize the distinction between the transition ID and the target state ID. When you refer to transitions, you will use the transition ID, not the target state ID.

Note that the trigger is not coded into the transition object. Your event handler must evaluate the trigger and then execute the transition. During runtime, if the trigger is met and your handler fires the transition, the manager tells state 1 to deactivate and then tells state 0 to activate. To execute the previous transition, your handler would issue the following statement.

```
s1.chg_st(2);
```

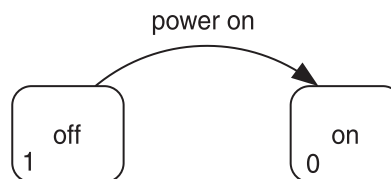


FIGURE 7–7

A simple transition, marked with a trigger.

2667-ch07 10/16/02 8:44 AM Page 85

Before issuing this statement, however, you must ensure that the source state is currently active. This can be accomplished by testing its *isActive* method, as follows.

```
If (s1.isActive())
    s1.chg_st(2);
```

Alternatively, you can see which state is active by examining the *cs* property of the state's manager (assuming *stMgr* is the state manager of *s1*), as follows.

```
if (stMgr.cs == 1) { ... }
```

Multi-level Transitions

In certain situations, you may need to transition from a source state to a target state that is not in the same level network as the source state. For example, in figure 7–8, the transition from state *x* to state *no power* is what we call a multi-level transition, or jump, because it transitions to a state other than a direct sibling.

Multi-level transitions should generally be avoided because too many of them may suggest that the grouping of contexts you have chosen for hierarchical states is not appropriate. However, multi-level transitions may not be avoidable and therefore can cross as many network boundaries as necessary.

Advanced: Using the State Manager to Effect Transitions

It may not be much of a secret, but when you want to change states, the state manager is centrally involved. So involved, in fact, that it is really the routine that handles the state changes. When you invoke the *chg_st* method of a state, the state manager is really doing the hard work.

So why should you care? You may find it more useful at times to invoke the transition with the state manager object, as in *stMgr.chg_st(0),* rather than invoking a transition with a state object, as in *s1.chg_st(0).* This statement tells the state manager to fire transition 0 in the current state, whatever that current state happens to be.

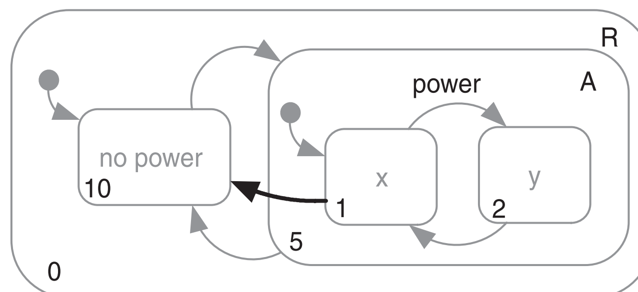


FIGURE 7–8

A multi-level transition from state *x* to the *no power* state.

2667-ch07 10/16/02 8:44 AM Page 86

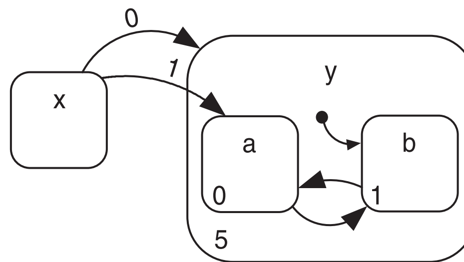


FIGURE 7–9
A multi-level transition from state *x* to the *no power* state.

In figure 7–9, you can see a multi-level transition (transition 1) used to jump into the hierarchical state *y*. Note that transition 0 and transition 1 are different. If transition 1 is triggered during runtime, state *x*'s state manager deactivates state *x*, activates state *y*, and then activates state *a*. For transition 0, however, state *x*'s state manager would deactivate state *x* and then activate state *y*. Then state *y*'s manager would find the default start state (*b*) and activate it.

NOTE: You can see examples of jumping several levels in the *TestCases.fla* file in the *FStEng* folder on the companion CD-ROM.

Coding multi-level transitions requires some manual effort because you need to figure out the common ancestor state of the source and target states, and explicitly code the path from the common ancestor to the target state. For example, the transition from state *x* in figure 7–8, if it had an ID of 0, would be coded as follows.

```
stX.add_trans(0, new trans_lvl(null, 1, [10]));
```

The first part, *add_trans,* is exactly the same as a simple transition. The second argument, however, is a *trans_lvl* object. The *trans_lvl* constructor takes three arguments. The first argument is a function to execute when the transition is fired, or null not to execute anything. The second argument is the number of levels to ascend in the state network to reach the common ancestor state. In figure 7–8, the common ancestor state is the root (managed by manager *R*). The third argument is an array specifying the states to activate, in order, from the ancestor level to the target state. The code for the multi-level transition in figure 7–9 is as follows.

```
stX.add_trans(1, new trans_lvl(null, 0, [5, 0]));
```

In this case, the common ancestor state is at the same level as the source state, so the manager does not have to ascend any levels. However, once it transitions to state 5 (*y*), it has to tell *y*'s manager to transition into state 0 (*a*), overriding the default start state mechanism.

Advanced: Transitioning into Concurrent States

In certain circumstances, you may need to transition into a substate of a concurrent hierarchical state. In figure 7–10, transition 1 from state *x* needs to activate the on state, instead of the default (off) state.

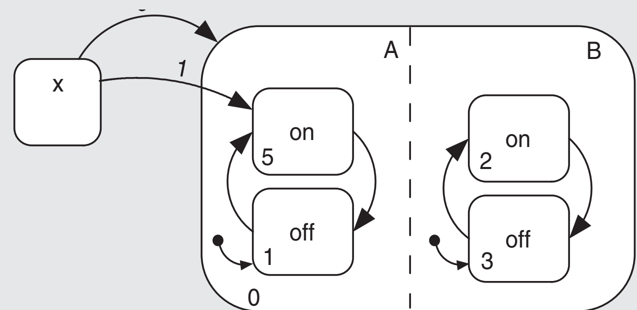
The code for the transition would look the same as entering a simple hierarchical state, as follows.

```
stX.add_trans(1, new trans_lvl(null, 0, [0, 5]));
```

If you were to invoke transition 0 from state *x,* the managers within the concurrent hierarchical state would activate states 1 and 3 by default. However, when transition 1 is triggered, the manager sees not to ascend any levels and then picks off the first state to enter, state 0 (the concurrent state). It then activates that state, but it uses the next state in the array, 5, to override the default state only for the network that contains that state (talk about intelligence!). The other networks activate their default start states. This means that you cannot override multiple networks within a concurrent hierarchical state, but if you need that feature, feel free to implement it yourself!

FIGURE 7–10

A multi-level transition into a concurrent state.



NOTE: We anticipate creating tools soon to alleviate the manual requirements. Stay tuned to *www.FlashSim.com* for details. Remember that when you fire a transition, the parameter is the transition identifier and not the target state. Keeping track of the numeric identifiers is probably the most difficult aspect of coding the statechart manually.

The State History Mechanism

Harel's statechart methodology and notation provide for a mechanism that remembers which substate was last visited within each hierarchical state. This is called the history mechanism. In Harel's notation, the history mechanism is denoted as an H within a circle, kind of like a special type of state.

2667-ch07 10/16/02 8:44 AM Page 88

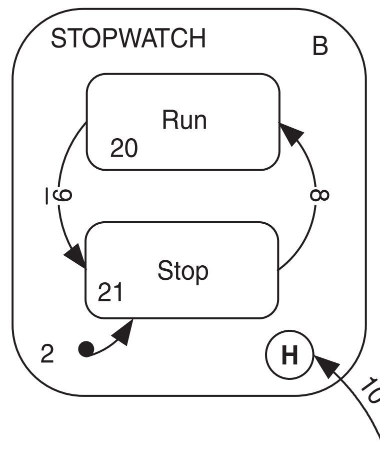


FIGURE 7–11
A history mechanism defined in state 2.

The history mechanism is useful for when the device has left a hierarchical state, done something else for a while, and now needs to get back to the hierarchical state and context. For example, in Chapter 16 we build a watch that has a stopwatch display, shown in figure 7–11. When you start the stopwatch and then leave the stopwatch display, and later you return, you want the stopwatch to be in the running stopwatch state.

By default, if you were to activate the stopwatch state (state 2), it would show the stopwatch stopped. However, when you follow the transition into the history mechanism, the state manager activates whichever state was active the last time the hierarchical state was active. Therefore, following transition 10, state manager B activates whichever state the display was in when the user left originally. If the history transition is used and the state has not been entered previously, the manager activates the default start state.

NOTE: The last state visited is kept in the *last_st* property of the state manager.

We actually lied a few pages ago when we said that simple transitions have two parameters and multi-level transitions have three parameters. In reality, they each have an optional final parameter, a true or false value indicating whether or not to use the history mechanism of the target state. If the target state is a simple state, this value is superfluous. If the final argument is not supplied, it defaults to false, meaning that the transition should not use the history mechanism.

You have to be careful about coding the history mechanism, because it can be deceptive. This history mechanism is like a state of the hierarchical state's network, and is therefore one level down from the hierarchical state. To code

2667-ch07 10/16/02 8:44 AM Page 89

Advanced: Clearing the History Settings

The history settings for each hierarchical state are stored in the state network managers. By default, when you activate the device, the state engine first clears all the history settings from all state managers. If for some reason you do not want to clear the history settings, you should activate the state engine, passing a true value as the optional argument, as follows.

```
myStateEngine.activate(true);
```

This tells the state engine not to reset the histories. If for any reason you want to reset the histories when the state engine is active, you can call the *reset_history()* method of any state manager. All states managed by that manager, and all state networks below, will have their history settings cleared.

the example in figure 7–11, you would therefore need a multi-level transition, as follows.

```
stSomeState.add_trans(10, new trans_lvl(null, 0, [2], true));
```

If you were to use a simple transition, as in the following, the manager would interpret this as meaning "go to the history at the current network level." Therefore, when the transition is fired, the manager would ignore the directive to go to state 2 and instead use the history mechanism to enter the most recent state visited (some sibling state of state 2).

```
stSomeState.add_trans(10, new trans (null, 2, true));
```

Transition to Self

A state transition can leave a state and return to the same state. This type of transition is called a transition to self and is denoted by a looping transition arrow from the source into itself, shown in figure 7–12. You are probably asking why this is necessary. The reason is that the activation and deactivation of states can trigger actions to occur, discussed in material to follow. Therefore, you may find it necessary, such as with a reset function, to leave (deactivate) the current state and then immediately enter (reactivate) the state.

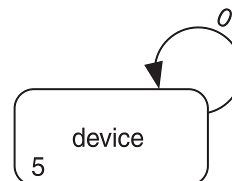


FIGURE 7–12
A simple transition to self.

2667-ch07 10/16/02 8:44 AM Page 90

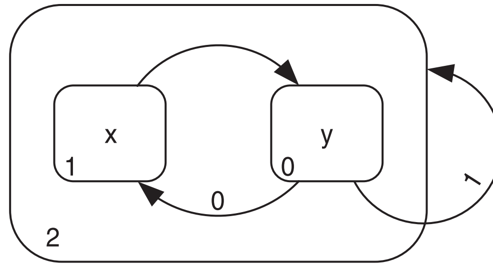


FIGURE 7–13
A multi-level transition to self.

The transition to self is typically a simple transition (figure 7–12) that follows the same pattern as other transitions, if it is written at the same network level. This is coded as follows.

```
device.add_trans(0, new trans(null, 5));
```

Rarely, the transition to self is a multi-level transition if you want to transition from a substate out of the hierarchical state and back into the hierarchical state (see figure 7–13).

This would be coded following the pattern of multi-level transitions, as in the following.

```
stY.add_trans(1, new trans_lvl(null, 1, [2]));
```

Advanced: Source-Indifferent Transitions

When you are designing your statecharts, particularly for interrupt-driven behaviors (i.e., on an interrupt, you want to go to a specific state, regardless of the current state), you may find that each state in a network needs a transition into the same target state under the same trigger. For example, in figure 7–14a, suppose that the trigger to get into state *x* from any other state (*m, y,* or *z*) is the same, such as "when the panic button is pressed." Rather than making separate transitions into *x* and cluttering the diagram, you can consolidate the transitions into what we call a single "source-indifferent transition," or transition 2 in figure 7–14b. From a conceptual point of view, this makes it easier to see that one enters state *x* regardless of the currently active substate.

In figure 7–14b, the designer has cleaned up the statechart shown in figure 7–14a by replacing state *y*'s transition 0, *z*'s transition 1, and *m*'s transition 0, with a single transition, 2, from the hierarchical state. Although the single transition now replaces three, the designer has possibly inadvertently added a side effect. If the network is already in state *x,* firing transition 2 will cause a transition to self (leave and then enter). This is different from figure 7-14a. Therefore, unless that behavior is intentional, you will need to add a condition to the trigger that rules out firing it if *x* is already active.

2667-ch07 10/16/02 8:44 AM Page 91

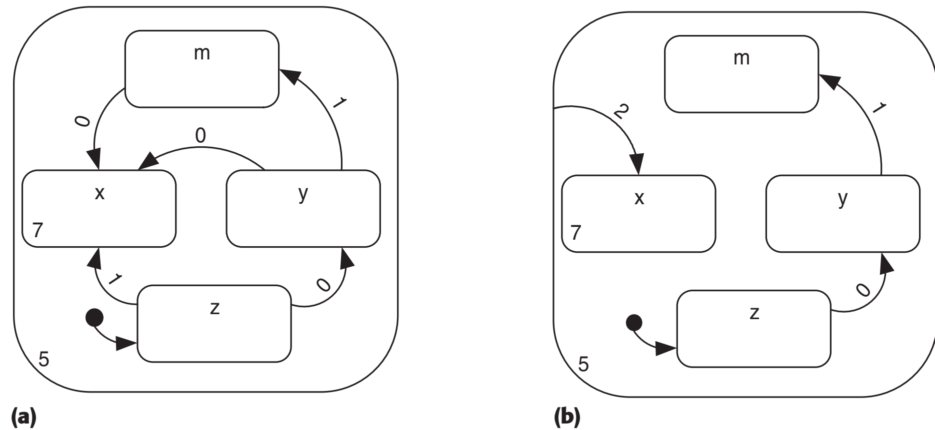


FIGURE 7–14
Defining a source-indifferent transition without the transition (a) and with the transition (b).

A source-indifferent transition is simply a generalization of a multi-level transition, and is coded as such. Because the source state can be any one of the sibling states, you attach the transition to the (parent) hierarchical state and descend, as opposed to ascend, into the first state along the path to the target, as follows.

```
st5.add_trans(2, new trans_lvl(null, -1, [7]));
```

In this example, you tell the manager to descend one level (–1) from state 5's network and transition into state 7 (*x*). If you wanted to go deeper, you would add more state IDs to the array, and if you wanted to use the history mechanism when you reached the target state's network, you would add *true* as a fourth argument to the *trans_lvl* constructor.

NOTE: Because this transition is attached to the parent state, you must ensure that the transition ID does not conflict with that state's other transitions.

ACTIONS AND ACTIVITIES

If the state network were only used to keep track of the current context, it would still be a very useful data structure. However, the real power of statecharts is used when you encode actions onto states and transitions. The actions attached to states and transitions implement the device's behavior as different contexts become active. In plain terms, you can add the following sets of actions.

- *Enter actions:* Actions that are triggered when a state is activated (entered).
- *Leave actions:* Actions that are triggered when a state first becomes deactivated (left).
- *Transition actions:* Actions that are triggered when a transition is fired, executed after the source state is deactivated but before the target state is activated.

2667-ch07 10/16/02 8:44 AM Page 92

- *Internal actions:* Actions that are triggered when you want to perform some actions but do not want to leave the current state.
- *Activate/deactivate actions:* Actions that are triggered when the state engine is activated or deactivated, respectively.

You can also add what we call *activities* to states; that is, actions that are invoked repeatedly while a state is active. There are two types of activities: pulse activities and continuous activities. This section reviews each type of action and demonstrates how to program it with the state engine implementation.

Enter Actions

Enter actions are actions that are executed immediately when a state becomes active. If the state is hierarchical, it first executes its enter actions and then activates the state manager(s), which activate its/their default start state(s). The process then repeats. In that way, the order of enter actions is completely predictable. This is useful on hierarchical states for defining code that should be executed regardless of the active subcontext.

In an object-oriented way, you define enter actions by overriding the default enter actions for the state. For example,

```
s13.entActs = function(){
    practice.sc.gotoAndStop(2);
    menubar.practice.gotoAndStop(1);
    menubar.review._visible = true;
}
```

Enter actions are contained in a newly created function. The actions can be any ActionScript instruction or expression. Within the function, you refer to elements on the current timeline with no prefix. If you need to refer to the state engine, you use the state engine variable name, as in the following.

```
se = new state_engine(. . .
...
s13.entActs = function () {
    se.someVal = 12;
    ...
```

NOTE: Be careful that you override the *entActs* method and not the *enter* method.

NOTE: If you need to refer to the state itself, such as if you want to fire a transition, you can use the *this* variable, as in *this.chg_st(0).* However, we strongly suggest that you evaluate triggers and invoke transitions within the event handler and not within the enter or leave actions. The one exception is that if you want to create what Harel calls a conditional state, which is a state that evaluates a condition and then immediately transitions to another state, you can use this mechanism.

2667-ch07 10/16/02 8:44 AM Page 93

You might want to reference the state engine if you need to perform computations on state engine properties you have created. Typically, you will create system "state variables"; that is, variables that are inherent to the system modeled in the state engine, as properties of the state engine as opposed to on the timeline, where they may interfere with other variables.

Leave Actions

Leave actions are actions that will execute just before a state becomes inactive. Leave actions are useful for cleaning up before the device leaves the current context. For example, if the device's power light goes on when you turn the device on, and off when you turn the device off, you might define the following actions.

```
sOn.entActs = function () {
    powLight.turnOn();
}

sOn.lvActs = function () {
    powLight.turnOff();
}
```

NOTE: Be careful that you override the *lvActs* method and not the *leave* method.

The state engine implementation guarantees that if you deactivate a substate its leave actions will execute before the leave actions of any parent or ancestor state. Therefore, *sOn* in the example could be a hierarchical state, and regardless of which state causes the transition out of *sOn* those actions will be executed in the expected order. The scope of leave actions is identical to that of enter actions, so you can refer to movie clips, properties, and so on exactly the same way.

Transition Actions

When a transition is fired, the context is switching in some way. You may want to perform some actions on that switch. Transition actions are the mechanism for doing this. Transition actions are executed after the source state has been deactivated and immediately before the target state is entered. In some special cases, knowing this order can be important to coding the correct actions.

Transition actions are added at the time you create the transition. The examples before did not define any transition action, so it was left as null. To code a transition action, you specify or define a function as the first parameter to the *trans* or *trans_lvl* constructor, as in the following.

```
afn = function () {
    trace("Transition actions from state 0 to 1");
}

s0.add_trans(1, new trans(afn, 2));
```

2667-ch07 10/16/02 8:44 AM Page 94

Advanced: Arguments to Transition Actions

Transition actions can be passed a single argument. Simply define a parameter for the function, and the function will receive the value if the transition is invoked with an argument. For example, the event handler could say the following.

```
s0.chg_st(1, someVal);
```

This will execute transition 1 on state 0, passing in the value *someVal* to your transition action function.

The last statement adds transition 1 to state 0, and when it is fired the state manager will invoke the *afn* function.

As far as scope is concerned, when you are coding transition functions, you can refer to the current timeline without qualifying it. However, the *this* variable points to the transition object, which is fairly useless for anything practical. If you need to refer to a state engine, you can use the state engine instance name.

Internal Actions

While a device is in a particular state, there may be several micro-behaviors you want it to perform without causing the network to change state. We call the mechanism "internal actions." Internal actions are triggered by events or conditions, and therefore the trigger is evaluated by the event handlers. For example, if you have a state with a stepper switch that increments or decrements an input value, it can be attached to an internal action on a state. When the state is active and the user presses the stepper switch, the state engine will execute the internal action that will increment the value. An example of an internal action is shown in figure 7–15.

Defining an internal action, as follows, is similar to defining a transition action.

```
s1.add_intActs(0, afn);
```

The first parameter is the internal action ID. Internal action IDs have to be unique only for the state. Like other IDs, internal action IDs are typically positive integers, but they do not have to be used sequentially. The second parameter is the function you want invoked when the internal actions are triggered. If the function is not on the current timeline, you will need to add the appropriate ActionScript prefixes. For obvious reasons, that function must be defined before you add the internal action with *add_intActs.*

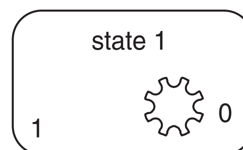


FIGURE 7–15
State 1 has an internal action with ID = 0.

Advanced: Arguments to Internal Actions

Like transitions, internal actions can accept an argument if one is specified when the internal action is invoked, as follows.

```
s1.exec_ia(0, someVal);
```

The internal action function must obviously be defined to accept the argument.

NOTE: Although the original statechart notation does not provide a symbol for internal actions, we have used a custom symbol on many of our statecharts for convenience, shown in figure 7–15.

To execute an internal action, the event handler uses the *exec_ia* method of the state, as follows.

```
if (s1.isActive() && trigger)
    s1.exec_ia(0);
```

Activating and Deactivating Actions

When you start up your device, you will activate the state engines you use for the control object and perhaps the model layer object. You can define actions to execute when the state engine is activated, overriding the *onActivate* method of the state engine, as follows.

```
se.onActivate = function () {
    // configure the device
}
```

You will typically put configuration code into the *onActivate* method, as demonstrated in the examples of Part IV. The default *onActivate* merely announces (using *trace*) that the state engine has been activated. The code inside *onActivate* has the scope of the state engine, and therefore references to *this* will reference the state engine data structure. The *onActivate* method is invoked before the root state is entered (activated), and the *onDeactivate* method is invoked after the root state is left (deactivated). Deactivation is similar in that you can override the *onDeactivate* method of the state engine.

NOTE: These actions are not part of Harel's notation, nor do we add a special symbol to a state network that indicates their presence, in that they are really outside the state network.

State Activities

Activities are sets of actions executed while a state is active, as opposed to just once when the state is entered or left, or once on transition or internal actions trigger. We define two types of activities: pulse activities, which are executed

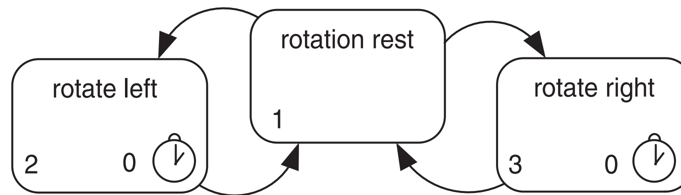


FIGURE 7–16
Pulse activities defined in rotate right and left states.

automatically at a preset frequency, and continuous activities, which operate continuously. Like enter and leave actions, you attach activities to the individual states.

Pulse Activities

With pulse activities, you set a desired frequency (in milliseconds) and a function or method to invoke. The state engine guarantees that when the state is active and the duration expires, the engine will invoke the function or method. For example, we could extend our robotic arm device (Chapter 15) to have a rotation rest state, a rotation left state, and a rotation right state, as shown in figure 7–16. When the rotation left is activated, a pulse activity attached to the state would cause the robotic arm to animate left at a predefined frequency. We have added pulse activities to the statechart notation as a stopwatch icon.

To implement pulse activities, the engine creates a timer (using Flash's *setInterval*), and when the time expires, the engine invokes the pulse activity. You can define as many pulse activities as you would like, but performance may suffer if you have many active at once and at very short intervals. Therefore, strive to limit the number of pulse activities, perhaps by consolidating functionality from several into one or few.

NOTE: Flash's setInterval is dependent on the frame rate for precise timing. Therefore, if you need very precise timing, you may want to use a timer component (explored in Chapter 12), set the resolution to be high (a low number), and turn it on in the state's enter actions and turn it off in the state's leave actions.

If you need a duration that is close to the frame rate, we recommend that you use a continuous activity instead (described in the following section).

There are two forms for adding pulse activities to states, which follow, differing in whether they invoke a function or method on the set frequency.

- *s1.add_pulse_activity(1, 100, myFn);*
- *s1.add_pulse_activity(0, 100, "myMethod", myObject);*

Both forms accept a pulse activity ID as the first argument, which must be unique among other pulse activities on the same state. The second parameter is

2667-ch07 10/16/02 8:44 AM Page 97

the interval to pulse, in milliseconds. The third argument, in the first form, is a function to call on each pulse. In the second form, the third and fourth argument specify a method and an object, respectively, to call on each pulse. Note that the method name is given within double quotation marks, but the object is given as a pointer. By using the second form, you can define the activities as methods of the state engine, to avoid interfering potentially with other symbols on the timeline. Within those methods, you can access other state engine properties using the *this* variable.

NOTE: When a state is activated, the pulse activity timer starts so that the first pulse will occur once the duration expires. If you want to invoke the actions immediately when the state is activated, you should call the function or method in your state's enter actions.

Continuous Activities

A continuous activity is an animation (movie clip) played while the state is active, and stopped when the state is deactivated. Its purpose is as an easy mechanism to start and stop an animation. Continuous activities are not part of Harel's notation, but we have added a symbol to represent it on the statechart (see figure 7–17).

You define your movie clip such that in frame 2 and beyond it performs the animation. If the animation will reach the end of the movie clip, you are responsible for telling the clip to *gotoAndPlay(2).* You will almost always have a stop action in frame 1, or else the movie clip will begin playing when it appears on the stage.

An option in defining continuous activities is whether or not to hide the movie clip when the state is inactive. If you create the activity setting this value to true, whenever the state is inactive the engine sets the clip's *_visible* property to false. When the state is activated, the engine sets the *_visible* property to true. When the state is deactivated, the engine sets the *_visible* property back to false. Continuous activities are added to states as follows.

```
s1.add_contin_activity(0, myAnimClip, true);
```

The first argument is an ID, which must be unique to other continuous activities attached to the state. The second argument is a pointer to the movie

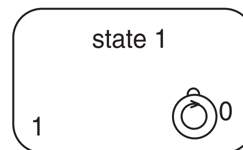


FIGURE 7–17

State 1 has a continuous activity.

clip (such as the movie clip's instance name, no quotation marks). The third argument says whether or not (true or false) to hide the clip when the state is inactive.

EVENT-ACTION TABLES

So far, you have seen how to design and implement states, transitions, and actions. The statechart is a visual representation of states and transitions, but there obviously is not space on the chart to include the details about triggers and actions. The *event-action* table is a textual table that details the triggers, actions, and activities you will program into your simulator.

In the process of developing a simulator, you will typically create the statechart first. As you develop the required functionality for each state, you will write out the triggers, actions, and activities in the event-action table. You will also come up with the different events and requests the state engine needs to process and generate, such as for the communication between the interface elements and the control object, and the control object and the model layer. Implementing the simulator is usually a straightforward, direct translation of the table into code, involving the following three tasks.

- Use the event-action table to determine which events and requests your event and request handlers need to process.
- Use the event-action table to design and code the event and request handlers.
- Decide how to implement the state actions and activities in ActionScript.

Some designers will include events and actions on their statecharts. We recommend, however, limiting statecharts to states and transitions and documenting events and actions on an accompanying event-action table. This separation allows you to keep cleaner-looking documentation while maintaining a complete description of the simulation for easy reference.

Elements of the Event-Action Table

You should create an event-action table for each state in your state engine. The event-action table may first be written in qualitative terms, and in more concrete terms as you refine your design. The event-action table should have the following headings, depending on whether you incorporate each item in the state.

- *Title:* State name, ID, and default start state (if hierarchical).
- Enter actions.
- Leave actions.
- *State activities:* This includes pulse and continuous activities. You should list each set of actions, along with frequency and other relevant parameters.
- *Internal actions:* The internal actions, which may be more than one set, should specify the trigger and the set of actions.

2667-ch07 10/16/02 8:44 AM Page 99

- *Transitions:* Each transition should have a trigger, a target state (which may be up or down some number of levels, a sequence of states to enter along the target path, and whether the history is to be used or not). If you want to execute actions along the transition, they should be included in the entry in the table.

The section that follows explores coding triggers; namely, events and conditions you use to fire transitions and invoke internal actions. You will code triggers into your event and request handlers.

Remember from Chapter 6 that most events and requests will come from dependent elements, such as the interface elements sending messages to the control object, or the control object sending messages to the model layer object. These messages are handled by your internal event and request handlers (*ieh* and *irh,* respectively). You may also receive events and requests from external sources.

These messages are handled by the external message handlers. In your event-action table, you may want to consider a state-centric approach in which you list both types of events (internal and external) in the same table. However, when you implement your design, you need to remember which events or requests are internal and external, for coding in the respective handlers. Alternatively, you can create two tables: one for internal event and request handling, and one for external. In our examples later in the book, we typically combine the events into a state-centric table, but use identifiers such as *ev()* and *x_ev()* to distinguish whether an event is internal or external.

Triggers

You decide what constitutes sufficient cause to transition from one state to another. These conditions can be simple, such as based purely on receiving an event, or they can be more complex, such as receiving an event and checking the value of a system property. Coding your event handler should be pretty tedious work. It consists of simply figuring out which context is current and then within that context interpreting the current event message to decide if any triggers are met, then firing one or more transitions (such as *s1.chg_st(0)*) or invoking internal actions (such as *s1.exec_ia(0)*). If the transition or internal action requires a value, you can pass it to the actions as a second argument to *chg_st* or to *exec_ia.*

Event-Only Triggers

The most common type of trigger is an event. Events originate from user interface objects, such as button presses and switch-position changes, or from nonvisual objects such as timer tick events. Events are messages sent from objects to the control object with at most one argument. These events in combination with the optional argument indicate that some incident has occurred, as well as providing additional information about the event.

For instance, a joystick can be coded to generate messages (*msgid*) at any time the joystick changes position. In addition to the message that a change

occurred, it can send an argument (*val*) communicating the *x* and *y* coordinates of the joystick to the event handler, as follows.

```
if (msgid == "JStickChg"){
    if (val.x == oldPoint.x){
        crane_net.chg_st(0);
    } else if (val.x > oldPoint.x){
        crane_net.chg_st(1);
    } else if (val.x < oldPoint.x){
        crane_net.chg_st(2);
    }
}
```

Multiple Triggers

Some transitions will include multiple event triggers. A multiple event trigger is a trigger that can be caused by more than one event. If a device has an auto shut-off feature, a transition from the ON state to the OFF state can be triggered by a switch movement from on to off, or by a timeout event, as follows.

```
if (msgid == "off" or msgid == "shutdown"){
    simpleDevice_net.chg_st(0);
}
```

Compound Triggers

A compound trigger, such as that which follows, causes a transition based on an event occurring and some condition evaluating to true. If a condition within a compound trigger evaluates to false, the event is ignored and the transition does not occur.

```
if (msgid == 1 and val.bank == "a" and abacusctrl.currentState <> 0){
    abacus_net.chg_st(0);
}
```

Translating Your Event-Action Table to Code

When you complete writing all of your event-action tables for all states, you are ready to code. You begin by creating your state engine, as follows.

```
myStateEngine = new state_engine(0, "my Enging", this);
```

The first parameter is an ID for the state engine, the second is a string name for debugging, and the third is the timeline on which the state engine is defined. Next, define the root state as either simple, hierarchical, or concurrent hierarchical, as follows.

```
stateNet = new hstate("State Network", 0, null);
```

2667-ch07 10/16/02 8:44 AM Page 101

The topmost state has no state manager, and hence the null value passed in as the third argument. However, if the top state is hierarchical (which it usually is), you should define a state manager for its immediate network, as follows.

```
stateNetMgr = new state_mgr("a", "top mgr", 0, stateNet);
```

This assumes that the default start state has state ID 0. Next, connect the state network to the state engine, as follows.

```
myStateEngine.set_state_network(stateNet);
```

Now you are ready to program the state structure. We suggest first creating the states, then the transitions (along with their actions), then the state actions and activities, and finally the event handlers.

Define each necessary handler by overriding the default state engine handler (*ieh, irh, xeh,* or *xrh*), as in the following.

```
myStatEngine.ieh = function (id, val) {
}
```

Appendix B (on the companion CD-ROM) provides useful advice on typical ways of structuring handlers.

When you are finished, you should connect the layers by using *addListener* and *setOracle,* as suggested in the previous chapter, and then activate the state engine, as follows.

```
myStateEngine.activate();
```

SUMMARY

This chapter has introduced the statechart notation and the basic constructs for creating state engine designs. The symbols are few in number, but powerful and logical for describing device behavior.

RESOURCES

Ian Horrocks' excellent book, *Constructing the User Interface with Statecharts,* published in 1999 by Addison-Wesley, is a comprehensive introduction and review of the statechart notation. You may also be interested in reading David Harel's original paper on statecharts, which was published in the journal *Science of Computer Programming*, vol. 8 (1987), pp. 231–274. As of this writing, the paper was available electronically from David Harel's Web site, *http://www.wisdom.weizmann.ac.il/~harel.*