CHAPTER

11 MODELING EVENT-BASED BEHAVIOR WITH STATE MACHINES

This chapter describes how to use state machines to model the behavior of blocks as they respond to internal and external events.

11.1 OVERVIEW

State machines are typically used in SysML to describe the state-dependent behavior of a block throughout its lifecycle, which is defined in terms of its states and the transitions between them. A state machine for a block may start, for example, when it initiates power up, then transition through multiple states in response to different stimuli, and terminate when it completes power down. The state machine defines how the block's behavior changes as it transitions between different states and while the block is in different states. State machines in SysML can be used to describe a wide range of state-related behavior, from the behavior of a simple lamp switch to the complex modes of an advanced aircraft.

State machines are normally owned by blocks and execute within the context of an instance of that block, but a state machine can also be owned by a package. The behavior of a state machine is specified by a set of regions, each of which contains its own states. The states in any one region are exclusive; that is, when the region is active, exactly one of its substates is active. A region normally has an initial pseudostate, which is the place the region starts executing when it first becomes active. When a state is entered, an (optional) entry behavior (e.g., an activity) is executed. Similarly, an optional exit behavior is executed on exit. While in a state, a state machine can execute a do behavior. A region also normally has a final state that signifies that the region has completed. Change of state is effected by transitions that connect a source state to a target state. Transitions are defined by triggers, guards, and effects. The trigger indicates an event that can cause a transition from the source state, the guard is evaluated in order to test whether the transition is valid, and the effect is a behavior executed once the transition is triggered. Triggers may be based on a variety of events such as the expiration of a timer or the receipt of a signal by the state machine's owning object.

Operation calls on the owning block are also valid trigger events for transitions. Junction and choice pseudostates support the construction of compound transitions between states, with multiple guards and effects.

State machines in different blocks may interact with one another by either sending signals or invoking operations. For example, the state machine of one block can send a signal to another block as part of a transition effect or state behavior. The event corresponding to the receipt of this signal by the receiving block can trigger a state transition in its state machine. Similarly, a state machine in one block may call an operation on another block that causes an event that triggers a transition.

A Practical Guide to SysML. http://dx.doi.org/10.1016/B978-0-12-800202-5.00011-4

State hierarchies occur when a state contains its own regions. A state with just one region is the most common case and is called a composite state. A state with more than one region is called an orthogonal composite state. Finally, a kind of state called a submachine state may reference another state machine. To model state hierarchies effectively, additional constructs are needed. Fork and join pseudostates are needed to specify transitions into and out of orthogonal composite states. Entry and exit point pseudostates can be used to add connection points for transitions on the boundary of a state or state machine.

State machines may also specify constraints within states or on transitions. The constraints may specify equations that correspond to different behaviors or different levels of performance that must be true in different states.

State machines can be used with other behaviors. For example, a state machine can use an activity or other behavior to specify what happens within a state, on entry, on exit, or on transition between states. State machines can also be used within interactions (see Chapter 10, Section 10.7.3) and activities (see Chapter 9, Section 9.11.3) to constrain certain aspects of their behavior. The integration of the semantics of different kinds of behaviors is sometimes complex and should be used with care.

11.2 STATE MACHINE DIAGRAM

State machine diagrams are sometimes referred to as **state charts** or state diagrams, but the actual name in SysML is the state machine diagram. The complete diagram header for a state machine diagram is as follows:

```
stm [stateMachine] state machine name [diagram name]
```

The diagram kind for a state machine diagram is **stm**, and the model element kind is always `stateMachine`. Because of this, the model element kind in square brackets is usually elided.

Figure 11.1 shows many of the basic notational elements for describing state machines. It describes a state machine for an ACME *Surveillance System.* It starts in the *idle* state, runs through a series of states during its

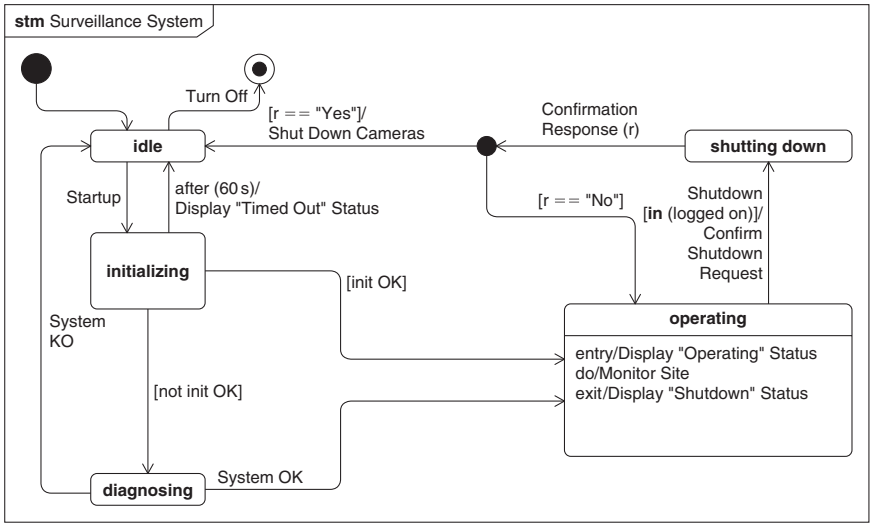

FIGURE 11.1

A state machine.

lifecycle, and finally ends up at *idle* again, when it may receive a *Turn Off* signal that causes it to complete its behavior. The notation for state machine diagrams is shown in the Appendix, Tables A.21 through A.23.

11.3 SPECIFYING STATES IN A STATE MACHINE

A **state machine** is a potentially reusable definition of some state-dependent behavior. State machines typically execute in the context of a block, and events experienced by the block instance may cause state transitions.

11.3.1 REGION

A state machine can contain one or more regions, which together describe the state-related behavior of the state machine. Each **region** is defined in terms of states and **pseudostates** and the transitions between them. An active region has exactly one active state within it at a given time. The difference between a state and a pseudostate is that a region can never stay in pseudostate, which merely exists to help determine the next active state. If a state machine contains a single region, it typically is not named, but if there are multiple regions, they are often named.

A state machine with multiple regions may describe some concurrent behavior happening within the state machine's owning block. This may represent an abstraction of the behavior of different parts of the block, as discussed in Chapter 7, Section 7.5.1. For example, one part of a factory may be storing incoming material, another turning raw material into finished products, and yet another sending out finished goods. The state machine may also include concurrent behaviors—such as a camera being panned and tilted at the same time—that are performed by multiple parts. If the parts' behaviors are specified, the relationship between the state machine for the parent block and the behaviors of its parts should also be specified. States can also contain multiple regions, as described in Section 11.6.2, but this section describes **simple states** only (i.e., states with no regions and therefore without nested states).

The initialization and completion of a region are described using an initial pseudostate and final state, respectively. An **initial pseudostate** is used to determine the initial state of a region. The outgoing transition from an initial pseudostate may include an effect (see Section 11.4.1 for a detailed discussion of transition effects). Such effects are often used to set the initial values of properties used by the state machine. When the active state of a region is the **final state**, the region has completed, and no more transitions take place within it. Hence, a final state can have no outgoing transitions.

The **terminate pseudostate** is always associated with the state of an entire state machine. If a terminate pseudostate is reached, then the behavior of the state machine terminates. A terminate pseudostate has the same effect as reaching the final states of all the state machine's regions. The termination of the state machine does not imply the destruction of its owning object, but it does mean that the object will not respond to events via its state machine.

If a state machine has a single region, it is represented by the area inside the frame of the state machine diagram. Multiple regions are shown separated by dashed lines.

The notation for the concepts introduced thus far is as follows:

- An initial pseudostate is shown as a filled circle.
- A final state is shown as a bulls-eye (i.e., a filled circle surrounded by a larger hollow circle).
- A terminate pseudostate is shown as an X.

11.3.2 STATE

A **state** represents some significant condition in the life of a block, typically because it represents some change in how the block responds to events and what behaviors it performs. This condition can be specified in terms of the values of selected properties of the block, but typically the condition is expressed in terms of implicit state variable(s) for each region. It is helpful to use the analogy that the block is controlled by a switch. Each state corresponds to a switch position for the block, and the block can exhibit some specified behavior in each switch position. The state machine defines all valid switch positions (i.e., states) and transitions between switch positions (i.e., state transitions). If there are multiple regions, each region is controlled by its own switch with its switch positions corresponding to its states. The switch positions can be specified by a form of truth table—similar to how logic gates can be specified—in which the current states and transitions define the next state.

Each state may contain **entry** and **exit behaviors** that are performed whenever the state is entered or exited, respectively. In addition, the state may contain a **do behavior** that executes once the entry behavior has completed. The do behavior continues to execute until it completes or the state is exited. Although any SysML behavior can be used, entry and exit behaviors and do behaviors are typically activities or opaque behaviors.

A state is represented by a round-cornered box containing its name. Entry and exit behaviors and do behaviors are described as text expressions preceded by the keywords `entry`, `exit`, or `do` and a forward slash. There is some flexibility in the content of the textual expression. The text expression typically is the name of the behavior, but when the behavior is an opaque behavior, the body of the opaque behavior can be used instead (refer to Chapter 7, Section 7.5 for a description of an opaque behavior).

Figure 11.2 shows a simple state machine for the *Surveillance System,* with a single *operating* state in its single region. A transition from the region's initial pseudostate goes to the *operating* state. On entry, the *Surveillance System* displays that it is operational on all operator consoles, and on exit, it displays a shutdown status. While the *Surveillance System* is in the *operating* state, it performs a `do` activity of its standard function to *Monitor Site*, which is monitoring the building where it is installed for any unauthorized entry. When in the *operating* state, a *Turn Off* signal triggers a transition to the final state, and because there is only a single region, the state machine terminates.

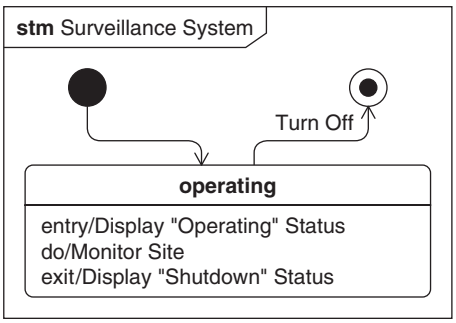

FIGURE 11.2

A state machine containing a single state.

11.4 TRANSITIONING BETWEEN STATES

A **transition** specifies when a change of state occurs within a state machine. State machines always run to completion once a transition is triggered, which means that they are not able to consume another trigger event until the state machine has completed the processing of the current event.

11.4.1 TRANSITION FUNDAMENTALS

A transition may include one or more triggers, a guard, and an effect as described next.

Trigger

A **trigger** identifies the possible stimuli that cause a transition to occur. SysML has four main kinds of triggering events.

- A **signal event** indicates that a new asynchronous message corresponding to a signal has arrived. A signal event may be accompanied by a number of arguments that can be used in the transition effect.
- A **time event** indicates either that a given time interval has passed since the current state was entered (relative) or that a given instant in time has been reached (absolute).
- A **change event** indicates that some condition has been satisfied (normally that some specific set of attribute values hold). Change events are discussed in Section 11.7.
- A **call event** indicates that an operation on the state machine's owning block has been requested. A call event may also be accompanied by a number of arguments. Call events are discussed in Section 11.5.

Once the entry behavior of a state has completed, transitions can be triggered by events irrespective of what is happening within the state. For example, a transition may be triggered while a do activity is executing, in which case the do activity is terminated.

By default, events must be consumed when they are presented to the state machine, even if they do not trigger transitions. However, events may be explicitly deferred while in a specific state for later handling. The deferred event is not consumed as long as the state machine remains in that state. As soon as the state machine enters a state in which the event is not deferred, the event must be consumed before any others. The event triggers a transition or it is consumed without any effect.

Transitions can also be triggered by internally generated **completion events.** For a simple state, a completion event is generated when the entry behavior and the do behavior have completed.

Guard

The **transition guard** contains an expression that must evaluate to true for the transition to occur. The guard is specified using a constraint, introduced in Chapter 8, Section 8.2, which includes a textual expression to represent the guard condition. When an event satisfies a trigger, the guard on the transition is evaluated. If the guard evaluates to true, the transition is triggered; if the guard evaluates to false, then the event is consumed with no effect. Guards can test the state of the state machine using the operators **in** (state x) and **not in** (state x).

Effect

The third part of the transition is the **transition effect.** The effect is a behavior, normally an activity or an opaque behavior, executed during the transition from one state to another. For a signal or call event, the arguments of the corresponding signal or operation call can either be used directly within the transition effect or be assigned to attributes of the block owning the state machine. The transition effect can be an arbitrarily complex behavior that may include send signal actions or operation calls used to interact with other blocks.

If the transition is triggered, first the exit behavior of the current (source) state is executed, then the transition effect is executed, and finally the entry behavior of the target state is executed.

A state machine can contain transitions, called internal transitions, which do not effect a change in state. An internal transition has the same source and destination and, if triggered, simply executes the transition effect. By contrast, an external transition with the same source and destination state—sometimes called a transition-to-self—triggers the execution of that state's entry and exit behaviors as well as the transition effect. One frequently overlooked consequence of internal transitions is that, because the state is not exited and entered, timers for relative time events are not reset.

Transition notation

A transition is shown as an arrow between two states, with the head pointing to the target state. Transitions-to-self are shown with both ends of the arrow attached to the same state. Internal transitions are not shown as graphical paths but are listed on separate lines within the state symbol, as shown in Figure 11.9.

The definition of the transition's behavior is shown in a formatted string on the transition with the list of triggers first, followed by a guard in square brackets, and finally the transition effect preceded by a forward slash. Section 11.4.3 describes an alternate graphical syntax for transitions.

The text for a trigger depends on the event, as follows:

- *Signal and call events*—the name of the signal or operation followed optionally by a list of attribute assignments in parentheses. Call events are typically distinguished by including the parentheses even when there are no attribute assignments. Although this is a useful convention, it is not part of the standard notation.
- *Time events*—the term `after` or `at` followed by the time. `after` indicates that the time is relative to the moment when the state is entered. `at` indicates that the time is an absolute time.
- *Change events*—the term `when` followed by the condition that has to be met in parentheses. Like other constraint expressions, the condition is expressed in text with the expression language optionally in braces.

The effect expression may either be the name of the invoked behavior or contain the text of an opaque behavior.

When an event is deferred in a state, the event is shown inside the state symbol using the text for the trigger followed by a "/" and the keyword defer. See Figure 11.12 for an example.

Transitions can also be named, in which case the name may appear alongside the transition instead of the transition expression. A name is sometimes a useful shorthand for a very long transition expression.

Figure 11.3 shows a more sophisticated state machine for the *Surveillance System* than in Figure 11.2, with all the principal states and the transitions between them. In contrast to Figure 11.2, the initial pseudostate now indicates that the region starts at the *idle* state. The final state is now also reached from the *idle* state, but it is still triggered by the receipt of a *Turn Off* signal. Once processing is complete in the *initializing* state (refer to Figure 11.14 to view inside the *initializing* state), a completion event for *initializing* will be

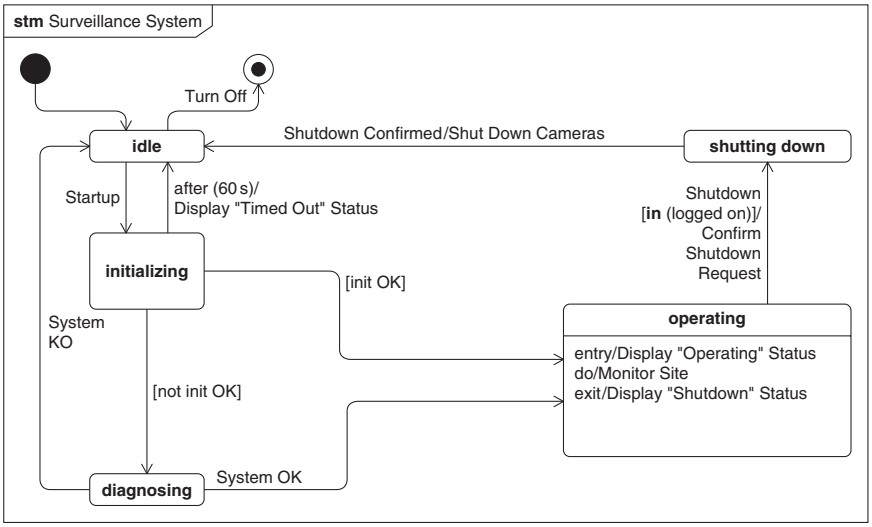

FIGURE 11.3

Transitions between states.

generated that triggers the two outgoing transitions. If the condition variable *init OK* is true, the system enters the *operating* state. Otherwise, the system enters the *diagnosing* state in which an operator will look at the error logs and try to manually initialize the system. Just in case something happens and the test procedure does not complete, the system has a time-out after 60 seconds, which returns the system to the *idle* state.

From the *diagnosing* state, the operator indicates success using the signal *System OK,* which allows the system to enter the *operating* state. The signal *System KO* indicates that the system is beyond operator repair and causes a transition back to *idle*. From the *operating* state, a *Shutdown* signal will cause a transition to the *shutting down* state, as long as the operating state is in substate *logged on* (refer to Figure 11.9 for a view inside the *operating* state). As part of shutting down, the system requests a confirmation and will only exit the *shutting down* state when it receives a *Shutdown Confirmed* signal, whereupon it executes the *Shut Down Cameras* activity.

Unless the graphical notation for transitions is being used (see Section 11.4.3), transition effect—with the exception of opaque behaviors—are specified on separate diagrams appropriate to the kind of behavior. Figure 11.4 shows the activity diagram for the *Shut Down Cameras* activity.

When invoked as a transition effect, *Shut Down Cameras* loops over all known cameras and sends each a *Shutdown* signal. Note that the activity does not include an accept event action; this would leave the invoking state machine in an ambiguous (mid-transition) state when waiting for new events to occur.

11.4.2 ROUTING TRANSITIONS USING PSEUDOSTATES

There are a variety of situations when a simple transition directly between two states is not sufficient to express the required semantics. SysML includes a number of pseudostates to provide these additional semantics. This section introduces junction and choice pseudostates, which support compound transitions between states.

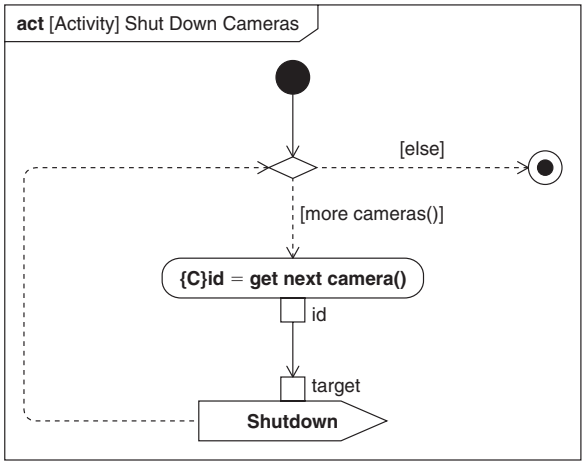

FIGURE 11.4

Defining a transition effect using an activity.

A **junction pseudostate** is used to construct a compound transition path between states. The compound transition allows more than one alternative transition path between states to be specified, although only one path can be taken in response to any single event. Multiple transitions may either converge on or diverge from the junction pseudostate. When there are multiple outgoing transitions from a junction pseudostate, the selected transition will be one of those whose guard evaluates to true at the time the triggering event is processed. If more than one guard evaluates to true, SysML does not define which one of the valid transitions is chosen for execution. If a particular compound transition path includes more than one junction between two states, all the guards along that path must evaluate to true before the compound transition is taken.

The **choice pseudostate** also has multiple incoming transitions and outgoing transitions and, like the junction pseudostate, is part of a compound transition between states. The behavior of the choice pseudostate is distinct from that of a junction pseudostate in that the guards on its outgoing transitions are not evaluated until the choice pseudostate has been reached. This allows effects executed on the prior transition to affect the outcome of the choice. When a choice pseudostate is reached in the execution of a state machine, there must always be at least one valid outgoing transition. If not, the state machine is invalid. A technique that is often used to ensure the validity of a choice pseudostate is to use a catch-all guard on no more than one outgoing transition. This is specified using the keyword `else`. Whether a compound transition contains junction pseudostates, choice pseudostates, or both, any possible compound transition must contain only one trigger, normally on the first transition in the path.

The various routing pseudostates are represented as follows:

- A junction pseudostate is shown, like an initial pseudostate, as a filled circle.
- A choice pseudostate is shown as a diamond.

Figure 11.5 completes the state machine for the *Surveillance System* shown in Figure 11.3. The handling of shutdown has been improved to describe what happens if the operator does not actually

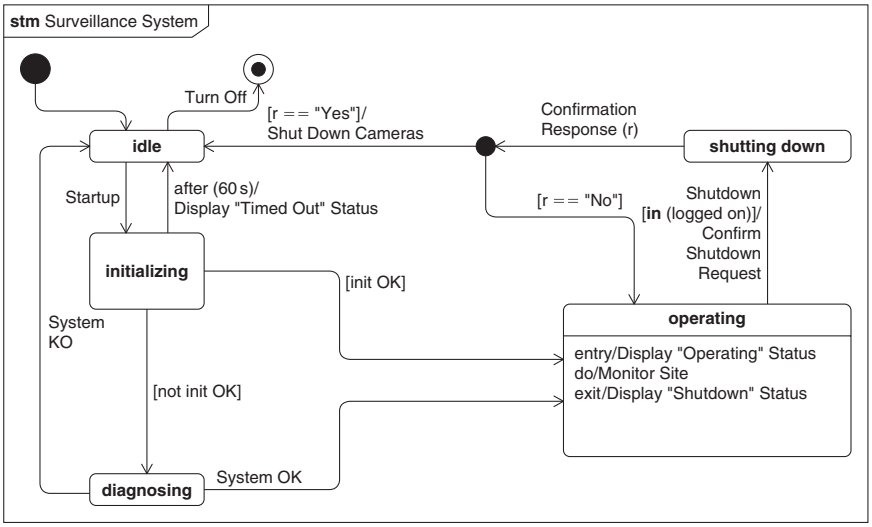

FIGURE 11.5

Routing transitions.

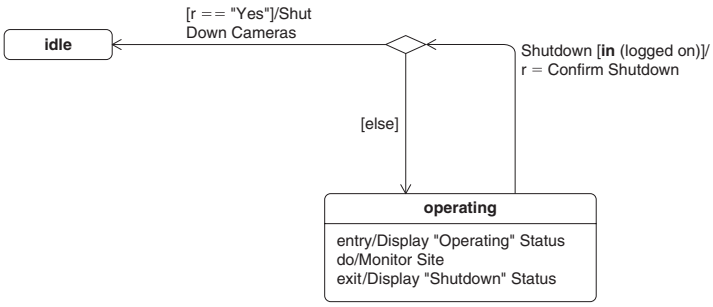

FIGURE 11.6

Specifying shutdown using a choice pseudostate.

want to shut down the system after all. The argument of the *Confirmation Response* signal, which takes values of *"Yes"* or *"No"* is mapped to attribute *r*. The transition triggered by the *Confirmation Response* signal now ends at a junction, with two outgoing transitions with different guards. If *r* == *"Yes"* then the system shutdown proceeds; if *r* == *"No,"* then the system returns to the operating state.

The transition from shutting down to idle/operating could be specified using a junction pseudostate in Figure 11.5 because the value of *r,* needed to determine the complete transition path, was available as part of the transition's trigger. However, Figure 11.6 shows another approach to system shutdown without a *shutting down* state. Here, the confirmation request is made as an effect of the transition out of the *operating* state, so the value of *r* is not known until after the first leg of the compound transition has been taken. In this case, a choice pseudostate is needed to allow the value of *r* returned from *Confirm Shutdown* to be used in the guard conditions on its exit transitions. As noted earlier, the modeler must ensure

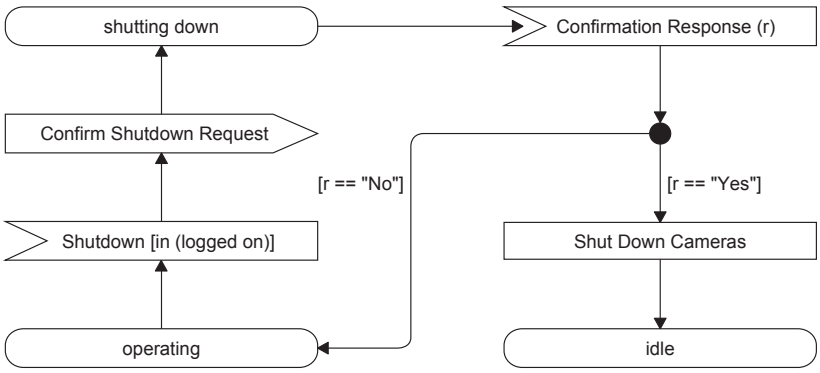

FIGURE 11.7

Transition-oriented notation.

that there is always at least one valid path from a choice pseudostate, so the guard on the transition has been changed to *[else]* in order to deal with any values other than *"Yes."* Then, even if *Confirm Shutdown* unexpectedly returns a value other than *"Yes"* or *"No,"* the state machine will still operate.

11.4.3 SHOWING TRANSITIONS GRAPHICALLY

Some modelers prefer to show transitions graphically on state machine diagrams. SysML introduces a set of special symbols that allow a modeler to depict send signal actions, other actions, and triggers graphically. These symbols are connected by arrows with solid heads to differentiate them from transition arrows. The graphical syntax for these symbols is as follows:

- A rectangle with a triangular notch removed from one side represents all the transition's triggers, with descriptions of the triggering events and the transition guard inside the symbol.
- A rectangle with a triangle attached to one side represents a send signal action. The signal's name, together with any arguments being sent, is shown within the symbol. There may be many send signal actions in a single transition effect, each with their own symbol. Signals are very important when communicating between state machines (hence the separate treatment of this action).
- Any other action in the transition effect is represented by a rectangle containing text that describes the action to be taken. There may be many actions as part of a transition effect, each with its own symbol.

Figure 11.7 shows the use of transition notation to provide an equivalent definition of the transitions between *operating, idle,* and *shutting down,* originally shown on Figure 11.5.

11.5 STATE MACHINES AND OPERATION CALLS

State machines can respond to operation calls on their parent block via call events. A call event may either be handled in a synchronous fashion—that is, the caller is blocked while waiting for a response—or asynchronously, which results in similar behavior to the receipt of a signal. The state machine executes all behaviors triggered by the call event until it has reached another state, and then returns any outputs created by those behaviors to the caller.

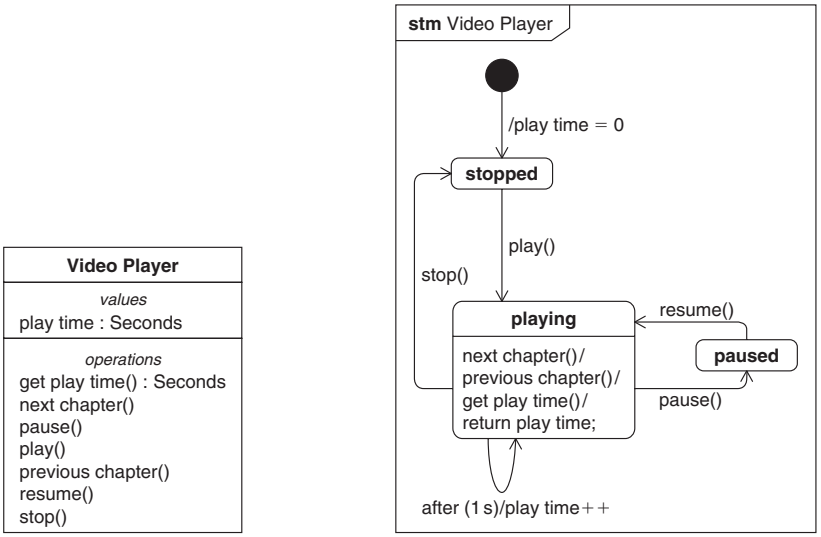

FIGURE 11.8

A state machine driven by call events for operations on its owning block.

One of the components used by the surveillance system's operators is a video player that allows them to review recorded surveillance data. The *Video Player* block, shown in Figure 11.8, provides a set of operations in its interface to control playback. Although many of the operations do not return data, it makes sense for any client of *Video Player* to wait until a request for these operations has been processed; hence, it makes sense for its interface to be defined using operations. The response of the block to requests from these operations is defined using the state machine shown in Figure 11.8, in which call events related to the operations are used as triggers on transitions. Calls to the *play, stop, pause,* and *resume* operations cause call events that trigger transitions between the various states of *Video Player.* Calls to the operations *next chapter, previous chapter,* and *get play time* cause call events that trigger internal transitions to state *playing.* To simplify the example, Figure 11.8 does not show many of the transition effects, but it does show how a request on *get play time* gets its return argument.

11.6 STATE HIERARCHIES

Just as state machines have regions, so can states; such states are called **composite** or **hierarchical states.** These allow state machines to scale to represent arbitrarily complex state-based behaviors. This section discusses composite states with single and multiple regions, as well as the reuse of an existing state machine to describe the behavior of a state.

11.6.1 COMPOSITE STATE WITH A SINGLE REGION

Arguably the most common situation is a composite state that has a single region. A state nested within the region can only be active when the state enclosing the region is active. Thus, the switch position

analogy described in Section 11.3.2 can apply to nested states by requiring that the switch position corresponding to the enclosing state be enabled in order to enable the switch positions corresponding to any of its nested states.

As stated earlier, a region typically will contain an initial pseudostate and a final state, a set of pseudostates, and set of substates, which may themselves be composite states. If the region has a final state, then a completion event is generated when that state is reached.

When an initial pseudostate is missing from a region in a composite state, the initial state of that region is undefined, although extensions to SysML are free to add their own semantics. However, a composite state may be porous, which means transitions may cross the state boundary, starting or ending on states within its regions (see Figure 11.10). In the case of a transition ending on a nested state, the entry behavior of the composite state, if any, is executed after the effect of the transition and before the execution of the entry behavior of the transition's target nested state. In the opposite case, the exit behavior of the composite state is executed after the exit behavior of the source nested state and before the transition effect. In the case of more deeply nested state hierarchies, the same rule can be applied recursively to all the composite states whose boundaries have been crossed.

Figure 11.9 shows the decomposition of the state *operating* from Figure 11.5 into the substates of one of its regions. On entry to the *operating* state, two entry behaviors are executed: the entry behavior of *operating, Display "Operating" status; logged in = 0,* and then the entry behavior of *logged off, Display "Logged Off."* This is because on entry, as indicated by the initial pseudostate, the initial substate of *operating* is *logged off.*

When in state *logged off,* a *Login* signal will cause a transition to the *logged on* state and will increment the value of *logged in.* While in the *logged on* state, repeated *Login* and *Logout* signals will increment and decrement the value of *logged in,* often as internal transitions without a change of state. However, if a *Logout* signal is received when the value of *logged in* is 1, then the signal will trigger a transition back to *logged off.* The entry behavior for *logged on* records the time in the variable *time on,* and its exit behavior uses that to display the *Session Length*.

The do activity *Monitor Site* executes as long as the state machine for the *Surveillance System* is in the *operating* state or until it reaches its own activity final. State *operating* does not have a final

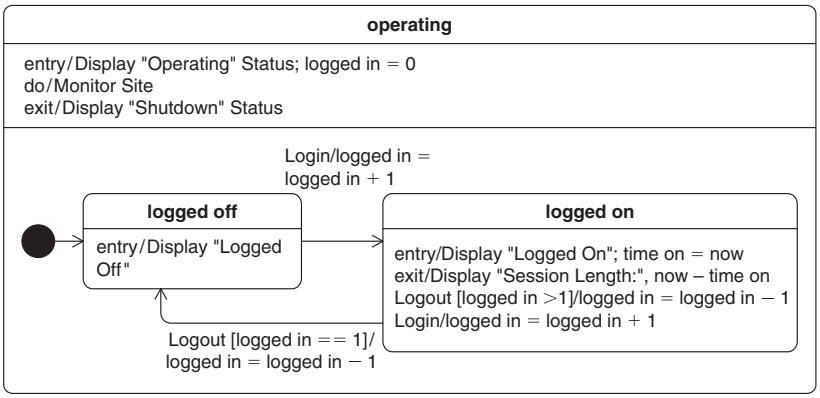

FIGURE 11.9

States nested within a composite state.

state, and so a completion event is never generated (as described above). As can be seen in Figure 11.5, this state is exited when a *Shutdown* signal is presented.

11.6.2 COMPOSITE STATE WITH MULTIPLE (ORTHOGONAL) REGIONS

A composite state may have many regions, which may each contain substates. A composite state with more than one region is sometimes called an **orthogonal composite state.** When an orthogonal composite state is active, each region has its own active state that is independent of the others, and any incoming event is independently analyzed within each region. A transition that ends on the composite state will trigger transitions from the initial pseudostate of each region, so there must be an initial pseudostate in each region for such a transition to be valid. Similarly, a completion event for the composite state will occur when all the regions are in their final state.

When an event is associated with triggers in multiple orthogonal regions, the event may trigger a transition in each region, assuming the transition is valid based on the other usual criteria. A simple example of this scenario is shown later in Figure 11.11.

Note that a transition can never cross the boundary between two regions of the same composite state. Such a transition, if triggered, would leave one of the regions with no active state, which is not allowed.

In addition to transitions that start or end on the composite state, transitions from outside the composite state may start or end on the nested states of its regions. In this case, one state in each region must be the start or end of one of a coordinated set of transitions. This coordination is performed by a fork pseudostate in the case of incoming transitions and a join pseudostate for outgoing transitions.

A **fork pseudostate** has a single incoming transition and as many outgoing transitions as there are orthogonal regions in the target state. Unlike junction and choice pseudostates, all outgoing transitions of a fork are part of the compound transition. When an incoming transition is taken to the fork pseudostate, all the outgoing transitions are taken. Because all outgoing transitions of the fork pseudostate have to be taken, they may not have triggers or guards but may have effects.

The coordination of outgoing transitions from an orthogonal composite state is performed using a **join pseudostate** that has multiple incoming transitions and one outgoing transition. The rules on triggers and guards for join pseudostates are the opposite of those for fork pseudostates. Incoming transitions of the join pseudostate may not have triggers or a guard but may have an effect. The outgoing transition may have triggers, a guard, and an effect. When all the incoming transitions can be taken and the join's outgoing transition is valid, the compound transition can occur. Incoming transitions occur first followed by the outgoing transition.

A fork and join pseudostate is shown as a vertical or horizontal bar with transition edges either starting or ending on the bar. An example of this can be seen in Figure 11.10, which shows a possible decomposition of the *operating* state from Figure 11.5.

The presence of multiple regions within a composite state is indicated by multiple compartments within the state symbol, separated by dashed lines. The regions can optionally be named, in which case the name appears at the top of the corresponding compartment. All nodes within such a compartment are part of the same region. As an alternative to showing the name of a state in a compartment, its name can be placed in a tab attached to the outside of the state symbol. An example of this can be seen in Figure 11.11.

Figure 11.10 shows a further elaboration of the *operating* state shown in Figure 11.9. In this elaboration, the *logged on* state has two orthogonal regions. One region, called *alert management,* specifies

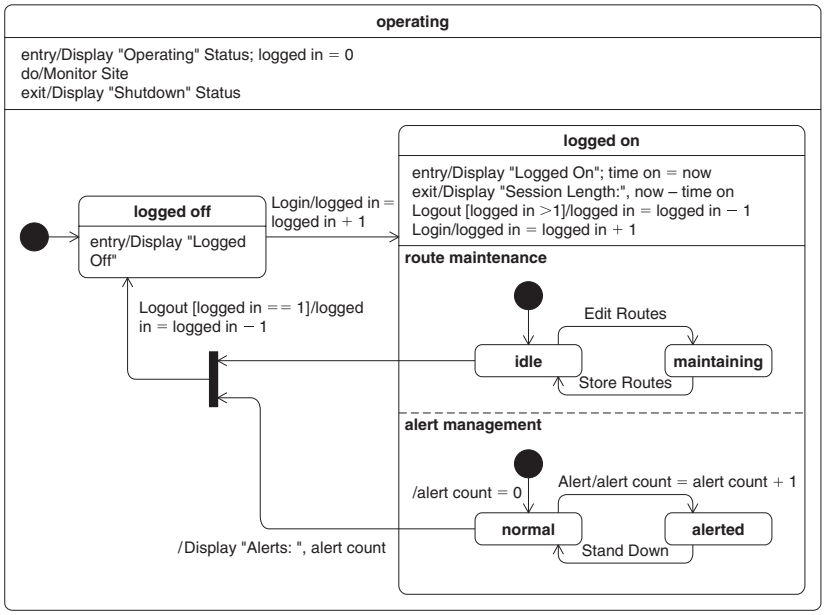

FIGURE 11.10

Entering and leaving a set of concurrent regions.

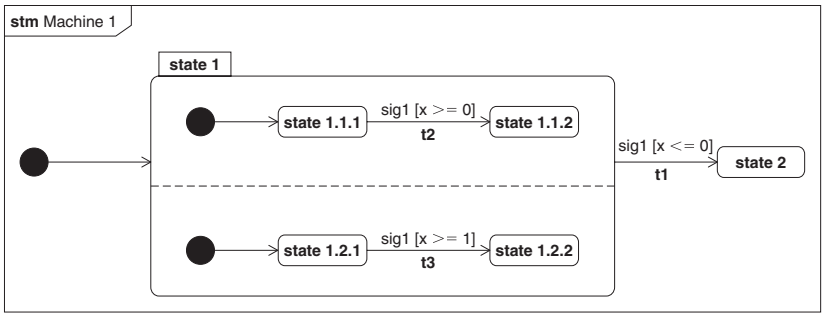

FIGURE 11.11

Illustration of transition firing order.

states and transitions for *normal* and *alerted* modes of operation; the other region, called *route maintenance,* specifies states and transitions for updating the route (i.e., pan-and-tilt angles) when the automatic surveillance feature of the system is engaged. As before, in state *logged off,* the receipt of a *Login* signal triggers transition to *logged on.* Based on the initial pseudostates in the two regions, the two initial substates of *logged on* are *idle* for region *route maintenance* and *normal* for region *alert management.* The receipt of an *Alert* signal triggers the transition from *normal* to *alerted* in *alert management.*

Similarly, the receipt of an *Edit Routes* signal triggers the transition from *idle* to *maintaining* in *route management.*

To ensure appropriate operator oversight of the system, the last operator can only log off if the *logged on* state is in substates *idle* and *normal.* This constraint is specified using a join pseudostate whose outgoing transition is triggered by a *Logout* signal with a guard of *logged in* = = *1.* The two incoming transitions to the join pseudostate start on *idle* and *normal,* so even if there is a *Logout* signal and the number of logged on operators is one, the outgoing transition from the join pseudostate will be valid only if the two active substates of *logged on* are *idle* and *normal.* Because the transitions from *idle* and *normal* cross the boundary of state *logged on,* its exit behavior is executed before any effects on the transitions. After evaluating that the guard condition on the transition evaluates to true, the order of execution triggered by the valid *Logout* signal is:

- exit behavior of *logged on—Display "Session Length:", now-time on*;
- incoming transition effect to join—*Display "Alerts:," alert count*;
- outgoing transition effect from join—*"logged in* = *logged in—1"*; and
- Entry behavior of logged off—*Display "Logged Off"*.

Having elaborated the *operating* state, it is apparent that the transitions *Logout [logged in* > *1]* and *Login* are rightly internal transitions rather than transitions-to-self. Transitions-to-self always exit and reenter the state, which in this case would reset the substates of *route maintenance* and *alert management*; obviously, this is not desirable in the middle of an intruder alert.

11.6.3 TRANSITION FIRING ORDER IN NESTED STATE HIERARCHIES

It is possible that the same event may trigger transitions at several levels in a state hierarchy, and with the exception of concurrent regions, only one of the transitions can be taken at a time. Priority is given to the transition whose source state is innermost in the state hierarchy.

Consider the state machine *Machine 1,* shown in Figure 11.11, in its initial state (i.e., in state 1.1.1 and 1.2.1). The signal *sig1* is associated with the triggers of three transitions, each with guards based on the value of variable *x.* Note that, in this case, the transitions have both a name and a transition expression, whereas a transition edge normally would show one or the other. This has been done to help explain the behavior of the state machine. The following list shows the transitions that will fire upon receipt of *sig1* based on values of *x* from −1 to 1:

- *x* equals −1—transition *t1* will be triggered because it is the only transition with a valid guard;
- *x* equals 0—transition *t2* will be triggered because, although transition *t**1*** also has a valid guard, *state 1.1.1* is the innermost of the two source states; or
- *x* equals 1—both transitions *t2* and *t3* will be triggered because both their guards are valid.

The normal rules for execution of exit behaviors apply, so, before the transition from *state 1* to *state 2* can be taken, any exit behavior of the active nested states of *state 1,* as well as the exit behavior *of state 1,* must be executed.

The example in Figure 11.11 is fairly straightforward. Assessing transition priority is more complex when compound transitions and transitions from within orthogonal composite states are used. However, the same rules apply.

11.6.4 USING THE HISTORY PSEUDOSTATE UPON RETURN TO A PREVIOUSLY INTERRUPTED REGION

In some design scenarios, it is desirable to handle an exception event by interrupting the behavior of the current region, responding to the event, and then returning back to the state that the region was in at the time of the interruption. This can be achieved by a kind of pseudostate called a **history pseudostate.** A history pseudostate represents the last active state of its owning region, and a transition ending on a history pseudostate has the effect of returning the region to that state. An outgoing transition from a history pseudostate designates a default history pseudostate. This is used when the region has no previous history or its last active state was a final state.

The two kinds of history pseudostate are deep and shallow. A **deep history pseudostate** records the states of all regions in the state hierarchy below and including the region that owns the deep history pseudostate. A **shallow history pseudostate** only records the top-level state of the region that owns it. As a result, the deep history pseudostate will enable a return to a nested state, while a shallow history pseudostate will enable a return to only the top-level state.

A history pseudostate is described using the letter "H" surrounded by a circle. The deep history pseudostate has a small asterisk in the top right corner of the circle.

The *Surveillance System* supports an emergency override mechanism, as shown in Figure 11.12. In a change from Figure 11.10, the reception of an *Override* signal with a valid password will always cause a transition from the *logged on* or *logged off* states, even if there is an ongoing alert. This transition is routed out of the enclosing *operating* state via an exit point pseudostate to the *emergency override activated* state (see a discussion of this at the end of Section 11.6.5). However, once the emergency is over, a *Resume Operation* signal needs to restore the *operating* state completely to its previous state so that the system can continue with its interrupted activities. To achieve this, the transition triggered

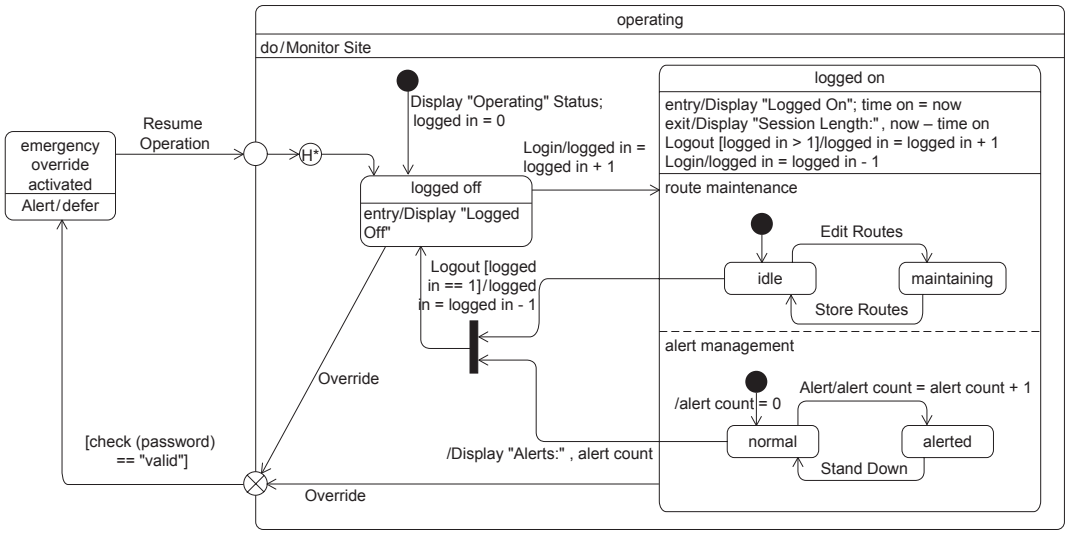

FIGURE 11.12

Recovering from an interruption using a history pseudostate.

by the *Resume Operation* signal ends (via an entry-point pseudostate) on a deep history pseudostate, which will restore the complete previous state of *operating*, including substates. By comparison, if a shallow history pseudostate was used, and the previous substate of *operating* was *logged on,* then the state machine would return to the initial states of logged on rather than previously active substates of *logged on.* If there is no previous history, the default state is *Logged Off.*

Alert events are deferred in the *emergency override activated* state so that they can be handled, if appropriate, in the resumed *operating* state.

11.6.5 REUSING STATE MACHINES

A **submachine state** is a kind of state that references a state machine that can be reused by other submachine states. A transition ending on a submachine state will start its referenced state machine. Similarly, when the referenced state machine completes, it will generate a completion event that can trigger transitions whose source is the submachine state. Modelers can also benefit from two additional kinds of pseudostates, called **entry** and **exit-point pseudostates**, which allow the state machine to define additional entry and exit points that can be accessed from a submachine state.

Entry and exit points on state machines

For a single-region state machine, entry- and exit-point pseudostates are similar to junctions; that is, they are part of a compound transition. Their outgoing guards have to be evaluated before the compound transition is triggered, and only one outgoing transition will be taken. On state machines, entry-point pseudostates can only have outgoing transitions, and exit-point pseudostates can only have incoming transitions.

Entry- and exit-point pseudostates are described by small circles that overlap the boundary of a state machine or composite state. An entry-point symbol is hollow, whereas an exit-point symbol contains an X.

Figure 11.13 shows a state machine for testing cameras, called *Test Camera,* which uses the graphical form for specifying transitions. From the entry-point pseudo state, the first transition simply sets the *failures* variable to 0 and ends on a choice pseudostate. On first entry, the state machine will always take the *[else]* transition, which will result in the sending of a *Test Camera* signal with the current camera number *(ccount)* as its argument. The state machine then stays in the *await test result* state until a *Test Complete* signal with argument *test result* has been received. The transition triggered by a *Test Complete* signal ends on a junction that either leads to the exit-point pseudostate *pass* (if the test passed) or back to the initial choice pseudostate (if the test failed), incrementing the *failures* variable on the way. If the camera has failed its self-test more than three times, then the transition with guard *[failures > 3]* will be taken to exit-point *fail.*

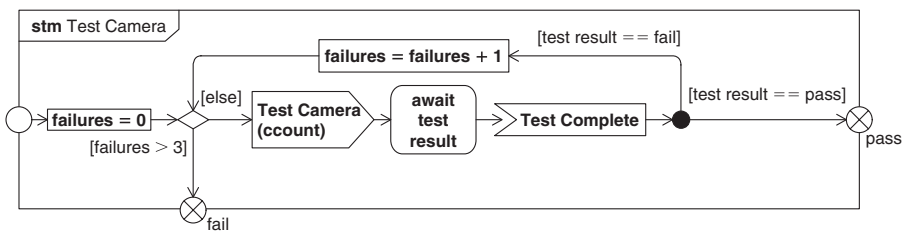

FIGURE 11.13

A state machine with entry and exit points.

Submachine states

A submachine state contains a reference to another state machine that is executed as part of the execution of the submachine state's parent. The entry- and exit-point pseudostates of the referenced state machine are represented on the boundary of the submachine state by special nodes called **connection points**. Connection points can be the source or target of transitions connected to states outside the submachine state. A transition whose source or target is a connection point forms part of a compound transition that includes the transition to or from the corresponding entry- and exit-point pseudostate in the referenced state machine. An example of this can be seen in Figure 11.14. In any given use of a state machine by a submachine state, only a subset of its entry and exit-point pseudostates may need to be externally connected.

A submachine state is represented by a state symbol showing the name of the state, along with the name of the referenced state machine, separated by a colon. A submachine state also includes an icon shown in the bottom right corner depicting either a simple state machine or a rake to be consistent with the representation of diagram decomposition in other diagrams. Connection points may be placed on the boundary of the submachine state symbol. These symbols are identical to the entry- and exit-point pseudostate symbols used in the referenced state machine. Note that only those connection points that need to be attached to transition edges need be shown on the diagram. Figure 11.14 shows the

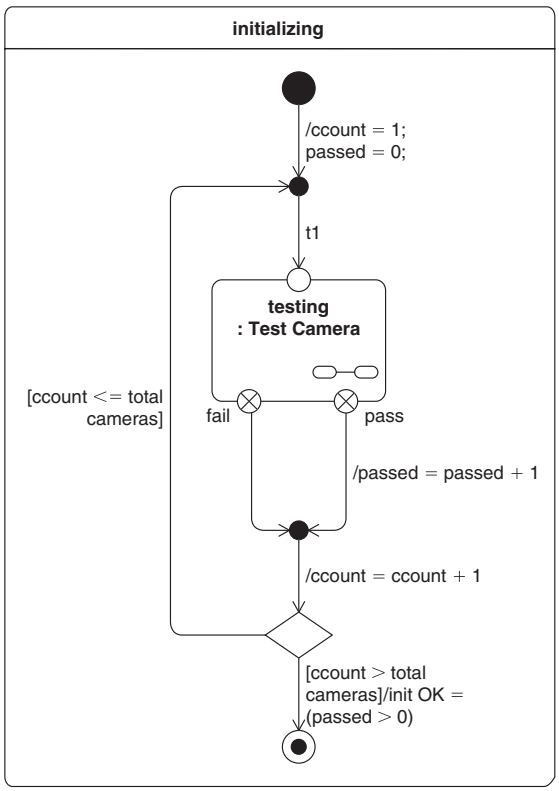

FIGURE 11.14

Invoking a substate machine.

initializing state of the *Surveillance System.* On entry, *ccount* (a property of the owning block that counts the number of cameras tested) and *passed* (a property that counts the number of cameras that passed their self-test) are initialized to 1 and 0, respectively. A junction pseudostate follows, which allows the algorithm to test as many cameras as required. To test each camera, the *testing* state uses the *Test Camera* state machine. The transition leaving the *pass* exit-point pseudostate has an effect that adds one to the *passed* variable; the transition leaving its *fail* exit-point pseudostate does not. Both transitions end in a junction whose outgoing transition increments the count of cameras tested. This transition ends on a choice, with one outgoing transition looping back to test another camera if *[ccount < = total cameras]* and the other reaching the final state of *initializing*. On the transition to the final state, the effect of the transition sets the *init OK* variable to true if at least one camera passed its self-test or false otherwise.

As stated earlier, entry- and exit-point pseudostates form part of a compound transition that, in the case of submachine states, incorporates transitions (and their triggers, guards, and effects) from both containing and referenced state machines. Looking at both Figure 11.13 and Figure 11.14, the compound transition from the initial pseudostate of state *initializing* will be as follows:

1. Initial pseudostate of the (single) region owned by state *initializing*
2. Transition labeled with effect *ccount = 1; passed = 0*
3. Transition named *t1*
4. Transition with effect *failures = 0*
5. Transition with guard *[else]* (at least this time)
6. (Graphical) transition with effect send *Test Camera* signal with argument *ccount*
7. State await test result.

Entry- and exit-point pseudostates on composite states

Entry-point and exit-point pseudostates can be used on the boundaries of composite states as well as a state machine. If the composite state has a single region, they behave like junctions. If the composite state has multiple regions, they behave like forks in the case of entry-point pseudostates and joins in the case of exit-point pseudostates. For entry-point pseudostates, the effects of their outgoing transitions execute after the entry behavior of the composite state. For exit-point pseudostates, their incoming transitions execute before the composite state's exit behavior. An example of entry-point and exit-point pseudodstates can be seen in Figure 11.12

11.7 CONTRASTING DISCRETE AND CONTINUOUS STATES

The examples shown so far in this chapter have been based on discrete semantics, specifically state machines in which the triggering event is a specific stimulus (i.e., a signal, an operation call, or the expiration of a timer). SysML state machines can also be used to describe systems with transitions that are driven by the values of either discrete or continuous properties. Such transitions are triggered by change events.

A trigger on a transition may be associated with a **change event** whose change expression states the conditions, typically in terms of the values of properties, which will cause the event to occur and hence trigger the transition. The change expression has a body containing the expression and an indication of the language used, which allows a wide variety of possible expressions.

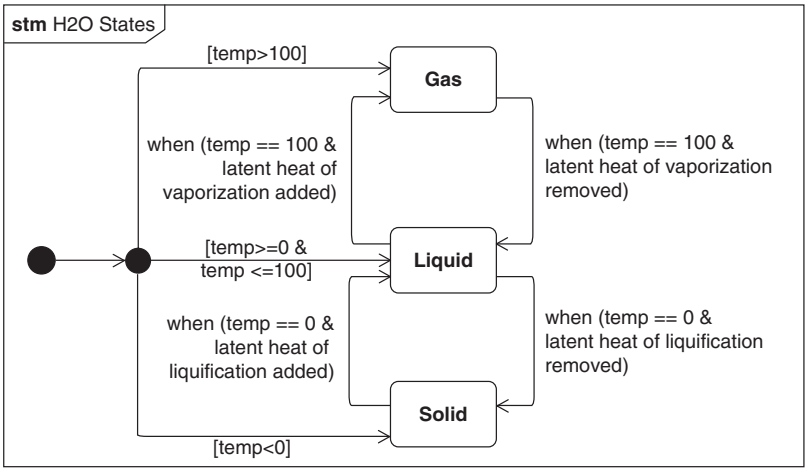

FIGURE 11.15

State machine for H_2O.

The state machine *H_2O States,* shown in Figure 11.15, defines the transitions between *Solid, Liquid,* and *Gas* states. These represent discrete states of *H_2O,* while the values of its properties, such as temperature and pressure, represent continuous state variables. Specific values for the variable *temp,* plus other conditions (e.g., the withdrawal or addition of energy), define the expressions for the change events and guards on the transitions. Implicitly, therefore, the values of its state variables are used to determine the discrete states of *H_2O* and the transitions between those states. Similarly, the discrete state of other continuous systems can be defined in terms of values of selected continuous properties of the system.

11.8 SUMMARY

A state machine is used to describe the behavior of a block in terms of its states and the transitions between them. State machines can be composed hierarchically like other SysML behavioral constructs, enabling arbitrarily complex representations of state-based behavior.

The significant state machine concepts covered in this chapter include the following:

- A state machine describes a potentially reusable definition of the state-dependent behavior of a block. Each state machine diagram describes a single state machine.
- Each state machine contains at least one region, which itself can contain a number of states and pseudostates, as well as the transitions between them. During execution of a state machine, each of its regions has a single active state that determines the transitions that are currently viable in that region. A region can have an initial pseudostate and final state that correspond to its beginning and completion, respectively.
- A state is an abstraction of some significant condition in the life of a block and specifies the effect of entering and leaving that condition and what the block does while it is in that condition using behaviors such as activities.

- Transitions describe valid state changes and under what circumstances those changes will happen. A transition has one or more triggers, a guard, and an effect. A trigger is associated with an event, which may correspond either to the reception of a signal (signal event) or operation call (call event) by the owning block; the expiration of a timer (time event); or the satisfaction of a condition specified in terms of properties of the block and its environment (change event). A transition can also be triggered by a completion event that occurs when the currently active state has completed.
- A guard expresses any additional constraints that need to be satisfied if the transition is to be triggered. If a valid event occurs, the guard is evaluated and, if true, the transition is triggered. Otherwise the event is consumed with no change in state. A transition can include a transition effect that is described by a behavior such as an activity. If the transition is triggered, the transition effect is executed.
- A state may specify that certain events can be deferred, in which case they are only consumed if they trigger a transition. Deferred events are consumed on transition to a state that does not further defer them.
- In a number of circumstances, simple transitions between states are not sufficient to specify the required behavior. Junction and choice pseudostates allow several transitions to be combined into a compound transition. Although the compound transition can include only one transition with triggers, it can have multiple transitions with guards and effects. Junction and choice pseudostates can have multiple incoming transitions and outgoing transitions. They are used to construct complex transitions that have more than one transition path, each potentially having its own guard and effect. History pseudostates allow a region to be interrupted and then subsequently to resume its previously active state or states.
- States may be composite with nested states in one or more regions. Just like state machines, during execution an active state will have one active substate per region. Composite states are porous; that is, transitions can cross their boundaries. Special pseudostates called fork and join pseudostates allow transitions to and from states in multiple regions at once. A given event may trigger transitions in multiple active regions.
- State machines may be reused via submachine states. Interactions with the reused state machine take place via transitions to and from the boundary of the corresponding submachine state, either directly or through entry- and exit-point pseudostates.
- Change events are driven by the values of variables of the state machine or properties of its owning block. In addition to discrete systems, change events can trigger transitions in continuous systems, in which transitions between the system's discrete states are triggered by changes in the values of continuous state variables. In this case, a behavior is a constraint on one or more state variables that must be true within a given state.

11.9 QUESTIONS

1. What is the diagram kind for a state machine diagram?
2. Which kinds of model element may a state machine region contain?
3. What is the difference between a state and a pseudostate?
4. A state machine has two states, "S1" and "S2." How do you show that the initial state for this machine is "S1"?

5. What is the difference between a final state and a terminate pseudostate?
6. A state has three behaviors associated with it. What are they called and when are they invoked?
7. What are the three components of a transition?
8. Under what circumstances does a completion event get generated for a state with a single region?
9. What is the difference in behavior between an internal transition and an external transition with the same source and target state?
10. What would the transition string for a transition look like if triggered by a signal event for signal "S1" with guard "a > 1" and an effect "a = a + 1"?
11. Draw the same transition using the graphical notation for transitions.
12. Where and how is a deferred event represented?
13. What is the difference between a junction and a choice pseudostate?
14. If a state has several orthogonal regions, how are they displayed?
15. What is the difference between a shallow and deep history pseudostate?
16. How can a state machine be reused within another state machine?
17. How are entry- and exit-point pseudostates represented on a state machine?
18. Under what circumstances will a given change event occur?

DISCUSSION TOPIC

State machines describe the behavior of blocks, but so do activities (via the use of activity partitions). Discuss approaches to ensuring that the two descriptions of behavior are consistent when both are used to describe the behavior of the same block.