Events and signals; State machine

In the real world, things happen. Not only do things happen, but lots of things may happen at the same time, and at the most unexpected times. "Things that happen" are called events, and each one represents the specification of a significant occurrence that has a location in time and space.

In the context of state machines, you use events to model the occurrence of a stimulus that can trigger a state transition. Events may include signals, calls, the passing of time, or a change in state.

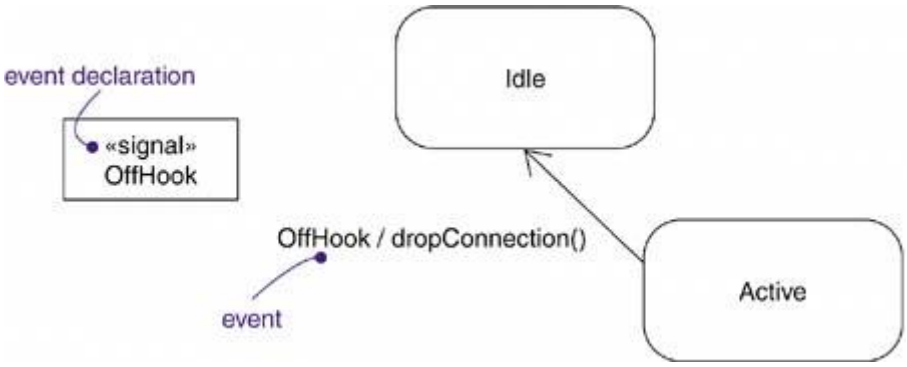
Events may be synchronous or asynchronous, so modeling events is wrapped up in the modeling of processes and threads.

A perfectly static system is intensely uninteresting because nothing ever happens. All real systems have some dynamic dimension to them, and these dynamics are triggered by things that happen externally or internally. At an ATM machine, action is initiated by a user pressing a button to start a transaction. In an autonomous robot, action is initiated by the robot bumping into an object. In a network router, action is initiated by the detection of an overflow of message buffers. In a chemical plant, action is initiated by the passage of time sufficient for a chemical reaction.

In the UML, each thing that happens is modeled as an event. An event is the specification of a significant occurrence that has a location in time and space. A signal, the passing of time, and a change of state are asynchronous events, representing events that can happen at arbitrary times. Calls are generally synchronous events, representing the invocation of an operation.

The UML provides a graphical representation of an event, as Figure 21-1 shows. This notation permits you to visualize the declaration of events (such as the signal `OffHook`) as well as the use of events to trigger a state transition (such as the signal `OffHook`, which causes a transition from the `Active` to the `Idle` state as well as the execution of the `dropConnection` action).

Figure 21-1. Events



Terms and Concepts

An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can

trigger a state transition. A *signal* is a kind of event that represents the specification of an asynchronous message communicated between instances.

Kinds of Events

Events may be external or internal. External events are those that pass between the system and its actors. For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

Signals

A message is a named object that is sent asynchronously by one object and then received by another. A signal is a classifier for messages; it is a message type.

Signals have a lot in common with plain classes. For example, signals may have instances, although you don't generally need to model them explicitly. Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general (for example, the signal `NetworkFailure`) and some of which are specific (for example, a specialization of `NetworkFailure` called `WarehouseServerFailure`). Also as for classes, signals may have attributes and operations. Before it has been sent by one object or after it is received by another, a signal is just an ordinary data object.

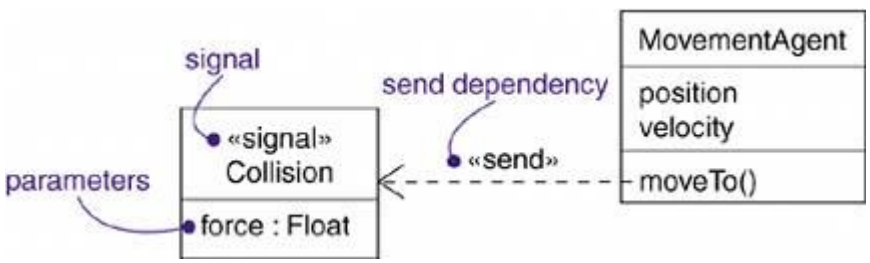
Note

The attributes of a signal serve as its parameters. For example, when you send a signal such as `Collision`, you can also specify a value for its attributes as parameters, such as `Collision(5.3)`.

A signal may be sent by the action of a transition in a state machine. It may be modeled as a message between two roles in an interaction. The execution of a method can also send signals. In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send.

In the UML, as Figure 21-2 shows, you model signals as stereotyped classes. You can use a dependency, stereotyped as `send`, to indicate that an operation sends a particular signal.

Figure 21-2. Signals



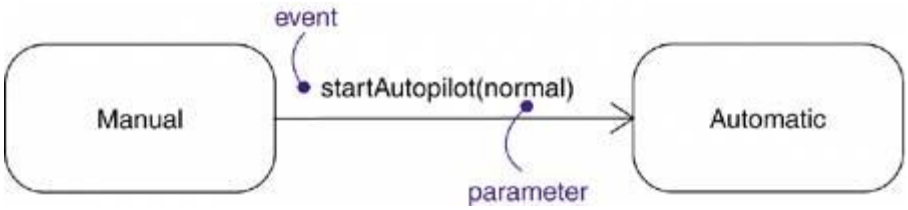
Call Events

Just as a signal event represents the occurrence of a signal, a call event represents the receipt by an object of a call request for an operation on the object. A call event may trigger a state transition in a state machine or it may invoke a method on the target object. The choice is specified in the class definition for the operation.

Whereas a signal is an asynchronous event, a call event is usually synchronous. This means that when an object invokes an operation on another object that has a state machine, control passes from the sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender. In those cases where the caller does not need to wait for a response, a call can be specified as asynchronous.

As Figure 21-3 shows, modeling a call event is indistinguishable from modeling a signal event. In both cases, you show the event, along with its parameters, as the trigger for a state transition.

Figure 21-3. Call Events



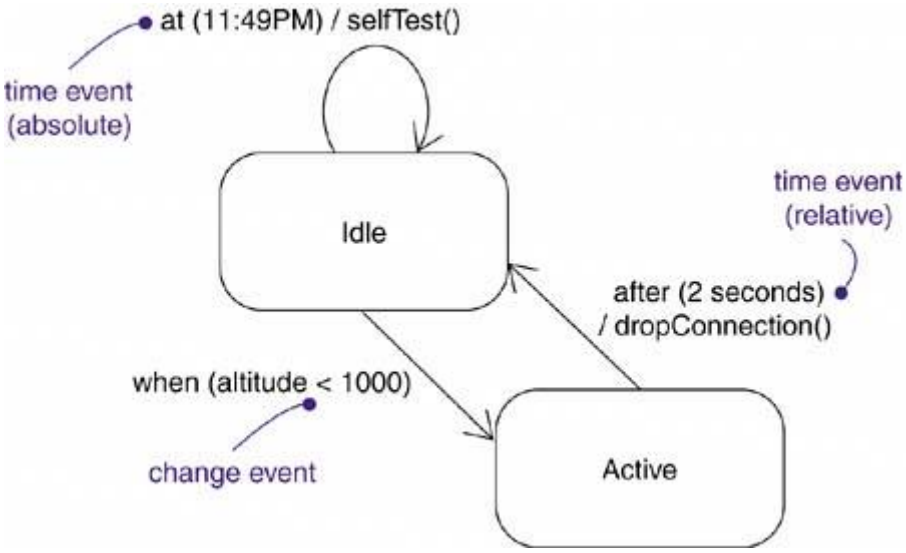
Note

Although there are no visual cues to distinguish a signal event from a call event, the difference is clear in the backplane of your model. The receiver of an event will know the difference, of course (by declaring the operation in its operation list). Typically, a signal will be handled by its state machine, and a call event will be handled by a method. You can use your tools to navigate from the event to the signal or the operation.

Time and Change Events

A time event is an event that represents the passage of time. As Figure 21-4 shows, in the UML you model a time event by using the keyword `after` followed by some expression that evaluates to a period of time. Such expressions can be simple (for example, `after 2 seconds`) or complex (for example, `after 1 ms since exiting Idle`). Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state. To indicate a time event that occurs at an absolute time, use the keyword `at`. For example, the time event `at (1 Jan 2005, 1200 UT)` specifies an event that occurs on noon Universal Time on New Year's Day 2005.

Figure 21-4. Time and Change Events



A change event is an event that represents a change in state or the satisfaction of some condition. As Figure 21-4 shows, in the UML you model a change event by using the keyword `when` followed by some Boolean expression. You can use such expressions for the continuous test of an expression (for example, `when altitude < 1000`).

A change event occurs once when the value of the condition changes from false to true. It does not occur when the value of the condition changes from true to false. The event does not recur while the event remains true.

Note

Although a change event models a condition that is tested continuously, you can typically analyze the situation to see when to test the condition at discrete points in time.

Sending and Receiving Events

Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation and the object to which the event is directed. Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active objects and passive objects.

Any instance of any class can send a signal to or invoke an operation of a receiving object. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver. For example, if an actor interacting with an ATM system sends the signal `pushButton`, the actor may continue along its way independent of the system to which the signal was sent. In contrast, when an object calls an operation, the sender dispatches the operation and then waits for the receiver to reply. For example, in a trading system, an instance of the class `trader` might invoke the operation `confirmTransaction` on some instance of the class `TRade`, thereby affecting the state of the `TRade` object. If this is a synchronous call, the `trader` object will wait until the operation is finished.

Note

In some situations, you may want to show one object sending a signal to a set of objects (multicasting) or to any object in the system that might be listening (broadcasting). To model multicasting, you'd show an object sending a signal to a collection containing a set of receivers. To model broadcasting, you'd show an object sending a signal to another object that represents the system as a whole.

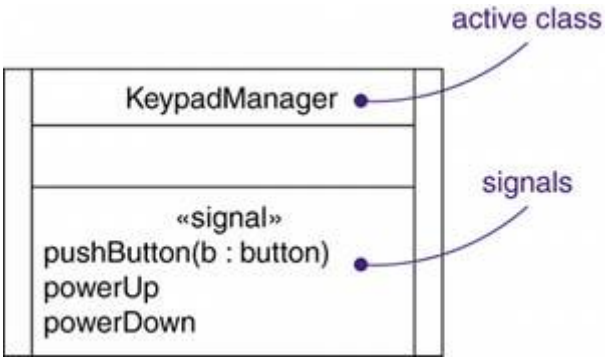
Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then the sender and the receiver are in a rendezvous for the duration of the operation. This means that the flow of control of the sender suspends until the execution of the operation completes. If this is a signal, then the sender and receiver do not rendezvous: The sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost (if no response to the event is specified), it may trigger the receiver's state machine (if there is one), or it may just invoke a normal method call.

Note

A call may be asynchronous. In this case, the caller continues immediately after issuing the call. The transmission of the message to the receiver and its execution by the receiver occur concurrently with the subsequent execution of the caller. When the execution of the method is complete, it just ends. If the method attempts to return values, they are ignored.

In the UML, you model the call events that an object may receive as operations on the class of the object. In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class, as shown in Figure 21-5.

Figure 21-5. Signals and Active Classes



Note

You can also attach named signals to an interface in this same manner. In either case, the signals you list in this extra compartment are not the declarations of a signal, but only the use of a signal.

Common Modeling Techniques

Modeling a Family of Signals

In most event-driven systems, signal events are hierarchical. For example, an autonomous robot might distinguish between external signals, such as a `Collision`, and internal ones, such as a `HardwareFault`. External and internal signals need not be disjoint,

however. Even within these two broad classifications, you might find specializations. For example, `HardwareFault` signals might be further specialized as `BatteryFault` and `MovementFault`. Even these might be further specialized, such as `MotorStall`, a kind of `MovementFault`.

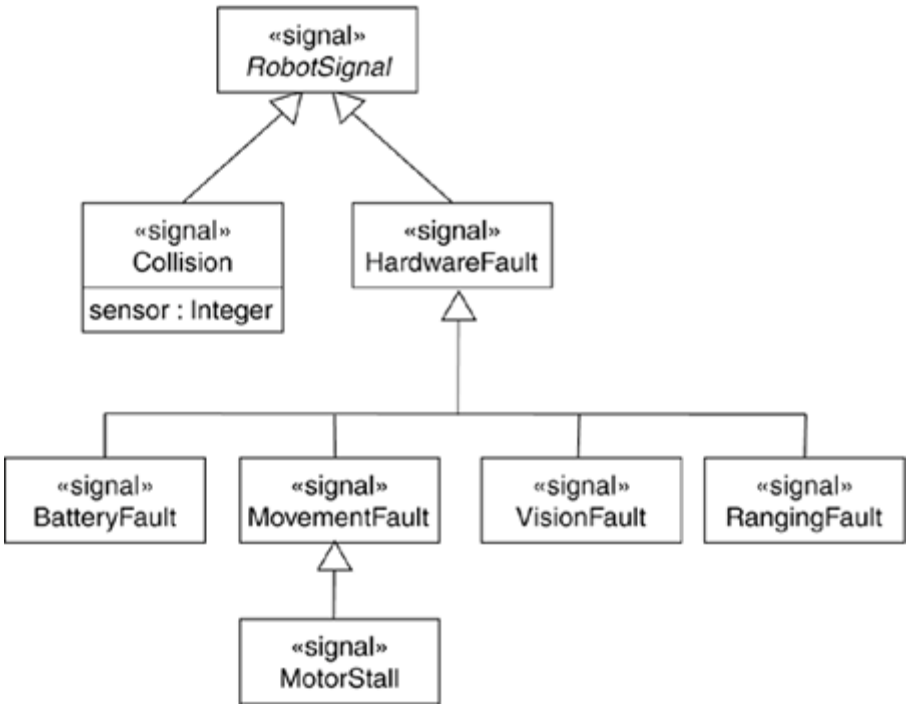
By modeling hierarchies of signals in this manner, you can specify polymorphic events. For example, consider a state machine with a transition triggered only by the receipt of a `MotorStall`. As a leaf signal in this hierarchy, the transition can be triggered only by that signal, so it is not polymorphic. In contrast, suppose you modeled the state machine with a transition triggered by the receipt of a `HardwareFault`. In this case, the transition is polymorphic and can be triggered by a `HardwareFault` or any of its specializations, including `BatteryFault`, `MovementFault`, and `MotorStall`.

To model a family of signals,

- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
- Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

Figure 21-6 models a family of signals that may be handled by an autonomous robot. Note that the root signal (`RobotSignal`) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (`Collision` and `HardwareFault`), one of which (`HardwareFault`) is further specialized. Note that the `Collision` signal has one parameter.

Figure 21-6. Modeling Families of Signals



Modeling Abnormal Occurrences

An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the abnormal occurrences that its operations can produce. If you are handed a class or an interface, the operations you can invoke will be clear, but the abnormal occurrences that each operation may raise will not be clear unless you model them explicitly.

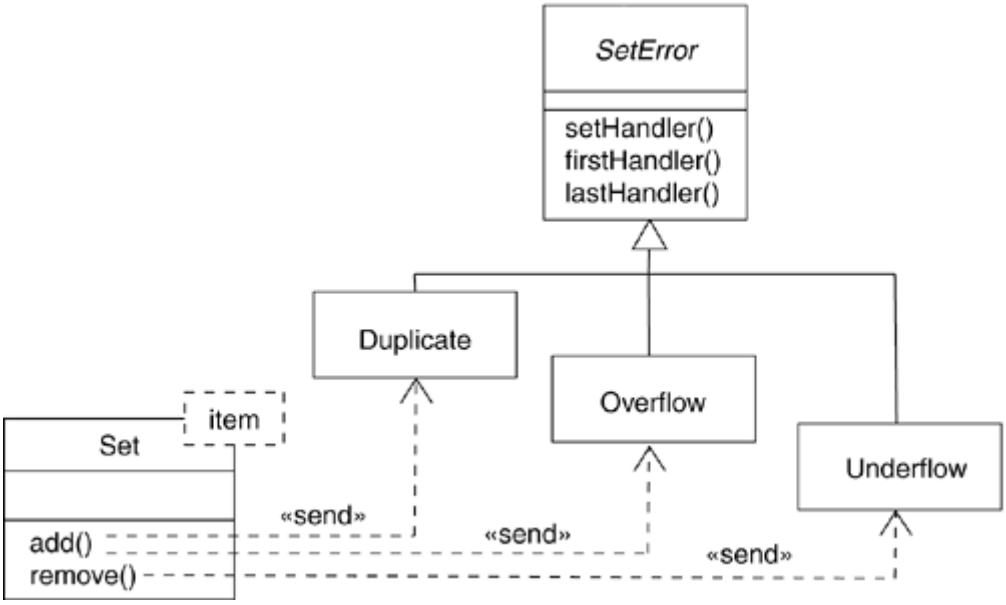
In the UML, abnormal occurrences are just additional kinds of events that can be modeled as signals. Error events may be attached to specification operations. Modeling exceptions is somewhat the inverse of modeling a general family of signals. You model a family of signals primarily to specify the kinds of signals an active object may receive; you model abnormal occurrences primarily to specify the kinds of abnormal occurrences that an object may produce.

To model abnormal occurrences

- For each class and interface, and for each operation of such elements, consider the normal things that happen. Then think of things that can go wrong and model them as signals among objects.
- Arrange the signals in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions as necessary.
- For each operation, specify the abnormal occurrence signals that it may raise. You can do so explicitly (by showing `send` dependencies from an operation to its signals) or you can use sequence diagrams illustrating various scenarios.

Figure 21-7 models a hierarchy of abnormal occurrences that may be produced by a standard library of container classes, such as the template class `Set`. This hierarchy is headed by the abstract signal `Error` and includes three specialized kinds of errors: `Duplicate`, `Overflow`, and `Underflow`. As shown, the `add` operation may produce `Duplicate` and `Overflow` signals, and the `remove` operation produces only the `Underflow` signal. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which signals each operation may send, you can create clients that use the `Set` class correctly.

Figure 21-7. Modeling Error Conditions



Note

Signals, including abnormal occurrence signals, are asynchronous events between objects. UML also includes exceptions such as those found in Ada or C++. Exceptions are conditions that cause the mainline execution path to be abandoned and a secondary execution path executed instead. Exceptions are not signals; instead, they are a convenient mechanism for specifying an alternate flow of control within a single synchronous thread of execution.

State Machines

Using an interaction, you can model the behavior of a society of objects that work together. Using a state machine, you can model the behavior of an individual object. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.

You use state machines to model the dynamic aspects of a system. For the most part, this involves specifying the lifetime of the instances of a class, a use case, or an entire system. These instances may respond to such events as signals, operations, or the passing of time. When an event occurs, some effect will take place, depending on the current state of the object. An *effect* is the specification of a behavior execution within a state machine. Effects ultimately resolve into in the execution of actions that change the state of an object or return values. A *state* of an object is a period of time during which it satisfies some condition, performs some activity, or waits for some event.

You can visualize the dynamics of execution in two ways: by emphasizing the flow of control from activity to activity (using activity diagrams) or by emphasizing the potential states of the objects and the transitions among those states (using state diagrams).

Well-structured state machines are like well-structured algorithms: They are efficient, simple, adaptable, and understandable.

Consider the life of your home's thermostat on one crisp fall day. In the wee hours of the morning, things are pretty quiet for the humble thermostat. The temperature of the house is stable and, save for a rogue gust of wind or a passing storm, the temperature

outside the house is stable, too. Toward dawn, however, things get more interesting. The sun starts to peek over the horizon, raising the ambient temperature slightly. Family members start to wake; someone might tumble out of bed and twist the thermostat's dial. Both of these events are significant to the home's heating and cooling system. The thermostat starts behaving like all good thermostats should, by commanding the home's heater to raise the inside temperature or the air conditioner to lower the inside temperature.

Once everyone has left for work or school, things get quiet, and the temperature of the house stabilizes once again. However, an automatic program might then cut in, commanding the thermostat to lower the temperature to save on electricity and gas. The thermostat goes back to work. Later in the day, the program comes alive again, this time commanding the thermostat to raise the temperature so that the family can come home to a cozy house.

In the evening, with the home filled with warm bodies and heat from cooking, the thermostat has a lot of work to do to keep the temperature even while it runs the heater and cooler efficiently. Finally, at night, things return to a quiet state.

A number of software-intensive systems behave just like that thermostat. A pacemaker runs continuously but adapts to changes in activity or heartbeat pattern. A network router runs continuously as well, silently guiding asynchronous streams of bits, sometimes adapting its behavior in response to commands from the network administrator. A cell phone works on demand, responding to input from the user and to messages from the local cells.

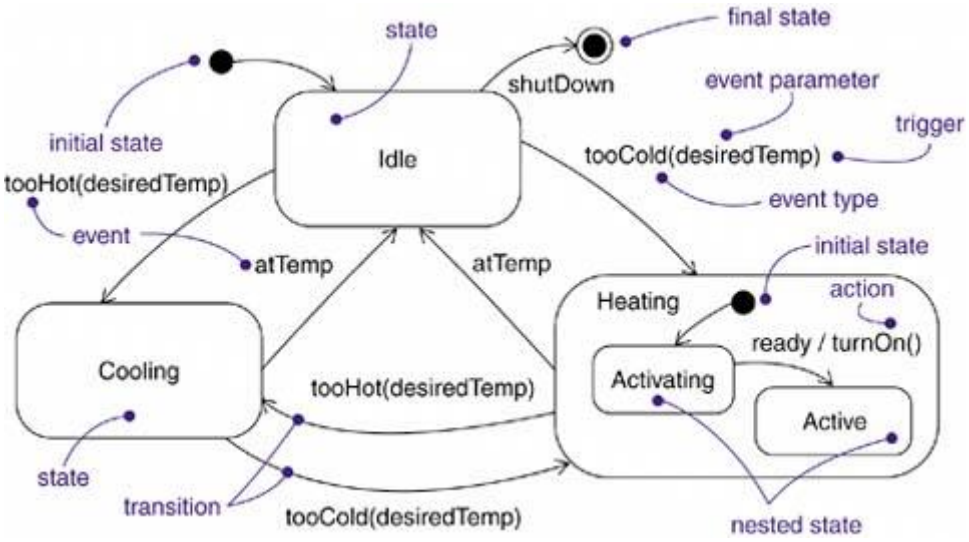
In the UML, you model the static aspects of a system by using such elements as class diagrams and object diagrams. These diagrams let you visualize, specify, construct, and document the things that live in your system, including classes, interfaces, components, nodes, and use cases and their instances, together with the way those things sit in relationship to one another.

In the UML, you model the dynamic aspects of a system by using state machines. Whereas an interaction models a society of objects that work together to carry out some action, a state machine models the lifetime of a single object, whether it is an instance of a class, a use case, or even an entire system. In the life of an object, it may be exposed to a variety of events, such as a signal, the invocation of an operation, the creation or destruction of the object, the passing of time, or the change in some condition. In response to these events, the object performs some action, which is a computation, and then it changes its state to a new value. The behavior of such an object is therefore affected by the past, at least as the past is reflected in the current state. An object may receive an event, respond with an action, then change its state. An object may receive another event and its response may be different, depending on its current state in response to the previous event.

You use state machines to model the behavior of any modeling element, most commonly a class, a use case, or an entire system. State machines may be visualized using state diagrams. You can focus on the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

The UML provides a graphical representation of states, transitions, events, and effects, as Figure 22-1 shows. This notation permits you to visualize the behavior of an object in a way that lets you emphasize the important elements in the life of that object.

Figure 22-1. State Machines



Terms and Concepts

A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for events. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* is ongoing nonatomic execution within a state machine. An *action* is an executable computation that results in a change in state of the model or the return of a value. Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line or path from the original state to the new state.

Context

Every object has a lifetime. On creation, an object is born; on destruction, an object ceases to exist. In between, an object may act on other objects (by sending them messages) as well as be acted on (by being the target of a message). In many cases, these messages will be simple, synchronous operation calls. For example, an instance of the class `Customer` might invoke the operation `getAccountBalance` on an instance of the class `BankAccount`. Objects such as these don't need a state machine to specify their behavior because their current behavior does not depend on their past.

In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous messages communicated between instances. For example, a cellular phone must respond to random phone calls (from other phones), keypad events (from the customer initiating a phone call), and to events from the network (when the phone moves from one call to another). Similarly, you'll encounter objects whose current behavior depends on their past behavior. For example, the behavior of an air-to-air missile guidance system will depend on its current state, such as `NotFlying` (it's not a good idea to launch a missile while it's attached to an aircraft that's still sitting on the ground)

or `Searching` (you shouldn't arm the missile until you have a good idea what it's going to hit).

The behavior of an object that must respond to asynchronous messages or whose current behavior depends on its past is best specified by using a state machine. This encompasses instances of classes that can receive signals, including many active objects. In fact, an object that receives a signal but has no transition for that signal in its current state and does not defer the signal in that state will simply ignore that signal. In other words, the absence of a transition for a signal is not an error; it means that the signal is not of interest at that point. You'll also use state machines to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

Note

Most of the time, you'll use interactions to model the behavior of a use case, but you can also use state machines for the same purpose. Similarly, you can use state machines to model the behavior of an interface. Although an interface may not have any direct instances, a class that realizes such an interface may. Such a class must conform to the behavior specified by the state machine of this interface.

States

A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time. For example, a `Heater` in a home might be in any of four states: `Idle` (waiting for a command to start heating the house), `Activating` (its gas is on, but it's waiting to come up to temperature), `Active` (its gas and blower are both on), and `ShuttingDown` (its gas is off but its blower is on, flushing residual heat from the system).

When an object's state machine is in a given state, the object is said to be in that state. For example, an instance of `Heater` might be `Idle` or perhaps `ShuttingDown`.

A state has several parts:

1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit effects	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving nonorthogonal (sequentially active) or orthogonal (concurrently active) substates
5. Deferred events	A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state

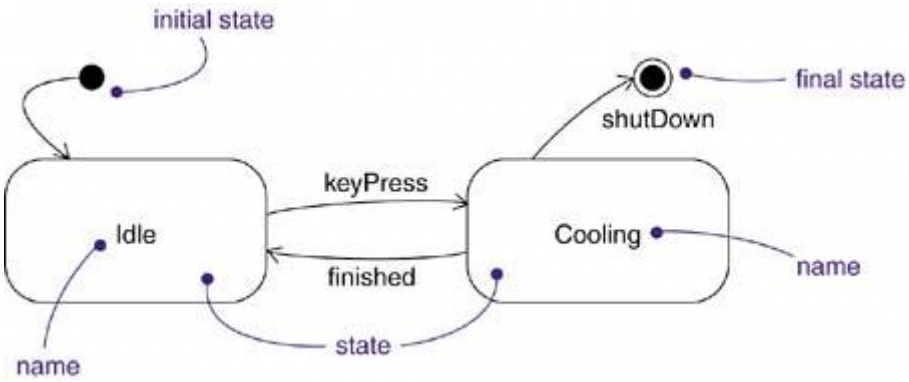
Note

A state name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon) and may continue over several lines. In practice, state names are short nouns or noun phrases drawn from the

vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a state name, as in `Idle` or `ShuttingDown`.

As Figure 22-2 shows, you represent a state as a rectangle with rounded corners.

Figure 22-2. States



Initial and Final States

As the figure shows, there are two special states that may be defined for an object's state machine. First, there's the initial state, which indicates the default starting place for the state machine or substate. An initial state is represented as a filled black circle. Second, there's the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle (a bull's eye).

Note

Initial and final states are really pseudostates. Neither may have the usual parts of a normal state, except for a name. A transition from an initial state to an ordinary state may have the full complement of features, including a guard condition and action (but not a trigger event).

Transitions

A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state. For example, a `Heater` might transition from the `Idle` to the `Activating` state when an event such as `tooCold` (with the parameter `desiredTemp`) occurs.

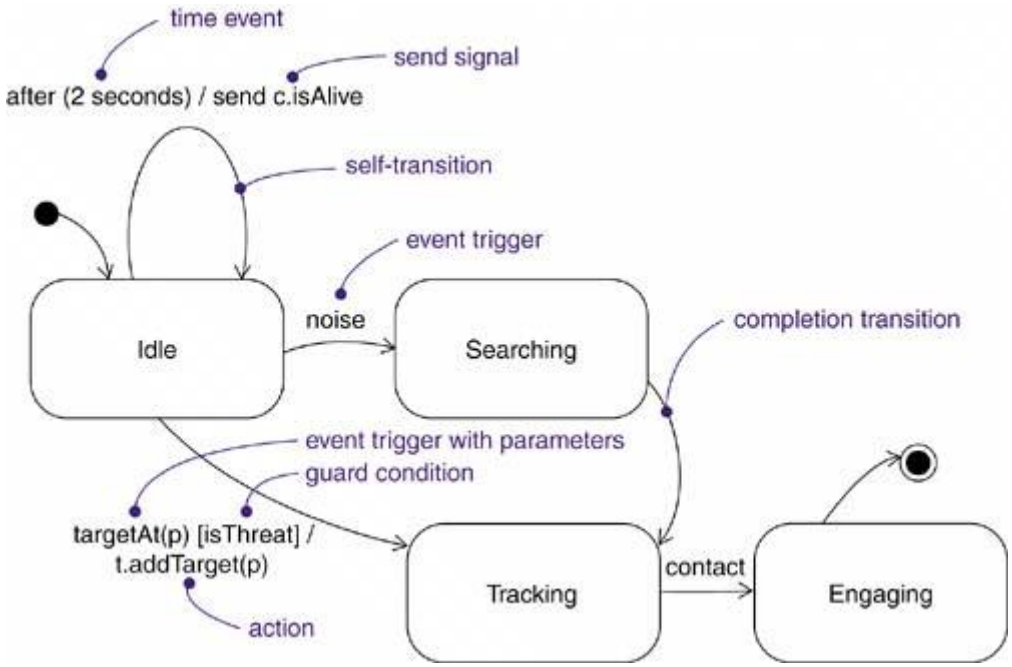
A transition has five parts.

1. Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied
2. Event trigger	The event whose recognition by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied

1. Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied
3. Guard condition	A Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates true, the transition is eligible to fire; if the expression evaluates false, the transition does not fire, and if there is no other transition that could be triggered by that same event, the event is lost
4. Effect	An executable behavior, such as an action, that may act on the object that owns the state machine and indirectly on other objects that are visible to the object
5. Target state	The state that is active after the completion of the transition

As Figure 22-3 shows, a transition is rendered as a solid directed line from the source to the target state. A self-transition is a transition whose source and target states are the same.

Figure 22-3. Transitions



Note

A transition may have multiple sources (in which case, it represents a join from multiple concurrent states) as well as multiple targets (in which case, it represents a fork to multiple concurrent states). See later discussion under orthogonal substates.

Event Trigger

An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. As shown in the previous figure, events may include signals,

calls, the passing of time, or a change in state. A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.

It is also possible to have a completion transition, represented by a transition with no event trigger. A completion transition is triggered implicitly when its source state has completed its behavior, if any.

Note

An event trigger may be polymorphic. For example, if you've specified a family of signals, then a transition whose trigger event is `S` can be triggered by `S`, as well as by any children of `S`.

Guard Condition

As the previous figure shows, a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event. A guard condition is evaluated only after the trigger event for its transition occurs. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.

A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered. Within the Boolean expression, you can include conditions about the state of an object (for example, the expression `aHeater in Idle`, which evaluates true if the `Heater` object is currently in the `Idle` state). If the condition is not true when it is tested, the event does not occur later when the condition becomes true. Use a change event to model that kind of behavior.

Note

Although a guard condition is evaluated only once each time its transition triggers, a change event is potentially evaluated continuously.

Effect

An effect is a behavior that is executed when a transition fires. Effects may include inline computation, operation calls (to the object that owns the state machine as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object. To indicate sending a signal you can prefix the signal name with the keyword `send` as a visual cue.

Transitions only occur when the state machine is quiescent, that is, when it is not executing an effect from a previous transition. The execution of the effect of a transition and any associated entry and exit effects run to completion before any additional events are allowed to cause additional transitions. This is in contrast to a do-activity (described later in this chapter), which may be interrupted by events.

Note

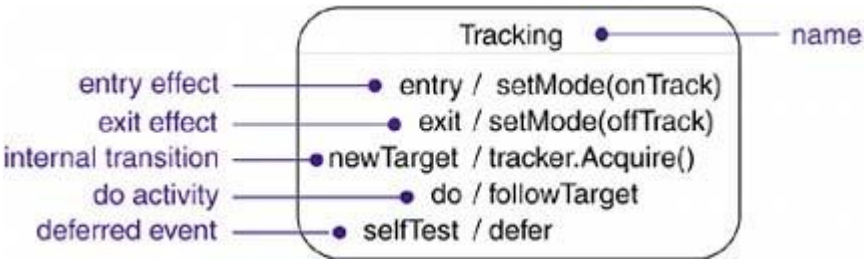
You can explicitly show the object to which a signal is sent by using a dependency stereotyped as `send`, whose source is the state and whose target is the object.

Advanced States and Transitions

You can model a wide variety of behavior using only the basic features of states and transitions in the UML. Using these features, you'll end up with flat state machines, which means that your behavioral models will consist of nothing more than arcs (transitions) and vertices (states).

UML state machines have a number of features that help you to manage complex behavioral models. These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines. Some of these advanced features include entry and exit effects, internal transitions, do-activities, and deferred events. These features are shown as text strings within a text compartment of the state symbol, as shown in Figure 22-4.

Figure 22-4. Advanced States and Transitions



Entry and Exit Effects

In a number of modeling situations, you'll want to perform some setup action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to perform some cleanup action no matter which transition led you away. For example, in a missile guidance system, you might want to explicitly announce the system is `onTrack` whenever it's in the `tracking` state, and `offTrack` whenever it's out of the state. Using flat state machines, you can achieve this effect by putting those actions on every entering and exiting transition, as appropriate. However, that's somewhat error prone; you have to remember to add these actions every time you add a new transition. Furthermore, modifying this action means that you have to touch every neighboring transition.

As Figure 22-4 shows, the UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry effect (marked by the keyword `entry`) and an exit effect (marked by the keyword `exit`), each with its appropriate action. Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

Entry and exit effects may not have arguments or guard conditions, but the entry effect at the top level of a state machine for a class may have parameters for the arguments that the machine receives when the object is created.

Internal Transitions

Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called internal transitions, and they are subtly different from self-transitions. In a self-transition, such as you see in Figure 21-3, an event triggers the transition, you leave the state, an action (if any) is performed, and then you reenter the same state. Because this transition exits and then enters the state, a self-transition executes the state's exit action, then it executes the action of the self-transition, and finally, it executes the state's entry action.

However, suppose you want to handle the event but don't want to execute the state's entry and exit actions. The UML provides a shorthand for this idiom using an internal transition. An *internal transition* is a transition that responds to an event by performing an effect but does not change state. In Figure 21-4, the event `newTarget` labels an internal transition; if this event occurs while the object is in the `tracking` state, action `tracker.acquire` is executed but the state remains the same, and no entry or exit actions are executed. You indicate an internal transition by including a transition string (including an event name, optional guard condition, and effect) inside the symbol for a state instead of on a transition arrow. Note that the keywords `entry`, `exit`, and `do` are reserved words that may not be used as event names. Whenever you are in the state and an event labeling an internal transition occurs, the corresponding effect is performed without leaving and then reentering the state. Therefore, the event is handled without invoking the state's exit and then entry actions.

Note

Internal transitions may have events with parameters and guard conditions.

Do-Activities

When an object is in a state, it generally sits idle, waiting for an event to occur. Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event. For example, if an object is in the `tracking` state, it might `followTarget` as long as it is in that state. As Figure 21-4 shows, in the UML you use the special `do` transition to specify the work that's to be done inside a state after the entry action is dispatched. You can also specify a behavior, such as a sequence of actionsfor example, do / op1(a); op2(b); op3(c). If the occurrence of an event causes a transition that forces an exit from the state, any ongoing do-activity of the state is immediately terminated.

Note

A do-activity is equivalent to an entry effect that starts the activity when the state is entered and an exit effect that stops the activity when the state is exited.

Deferred Events

Consider a state such as `tracking`. As illustrated in Figure 21-3, suppose there's only one transition leading out of this state, triggered by the event `contact`. While in the state `TRacking`, any events other than `contact` and other than those handled by its substates will be lost. That means that the event may occur, but it will be ignored and no action will result because of the presence of that event.

In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out. However, in some modeling situations, you'll want to accept some events but postpone a response to them until later. For example, while in

the `TRacking` state, you may want to postpone a response to signals such as `selfTest`, perhaps sent by some maintenance agent in the system.

In the UML, you can specify this behavior by using deferred events. A deferred event is an event whose processing in the state is postponed until another state becomes active; if the event is not deferred in that state, the event is handled and may trigger transitions as if it had just occurred. If the state machine passes through a sequence of states in which the event is deferred, it is preserved until a state is finally encountered in which the event is not deferred. Other nondeferred events may occur during the interval. As you can see in Figure 21-4, you can specify a deferred event by listing the event with the special action `defer`. In this example, `selfTest` events may happen while in the `TRacking` state, but they are held until the object is in the `Engaging` state, at which time it appears as if they just occurred.

Note

The implementation of deferred events requires the presence of an internal queue of events. If an event happens but is listed as deferred, it is queued. Events are taken off this queue as soon as the object enters a state that does not defer these events.

Submachines

A state machine may be referenced within another state machine. Such a referenced state machine is called a *submachine*. They are useful in building large state models in a structured manner. See the *UML Reference Manual* for details.

Substates

These advanced features of states and transitions solve a number of common state machine modeling problems. However, there's one more feature of the UML's state machinessubstatesthat does even more to help you simplify the modeling of complex behaviors. A substate is a state that's nested inside another one. For example, a `Heater` might be in the `Heating` state, but also while in the `Heating` state, there might be a nested state called `Activating`. In this case, it's proper to say that the object is both `Heating` and `Activating`.

A simple state is a state that has no substructure. A state that has substatesthat is, nested statesis called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (nonorthogonal) substates. In the UML, you render a composite state just as you do a simple state, but with an optional graphic compartment that shows a nested state machine. Substates may be nested to any level.

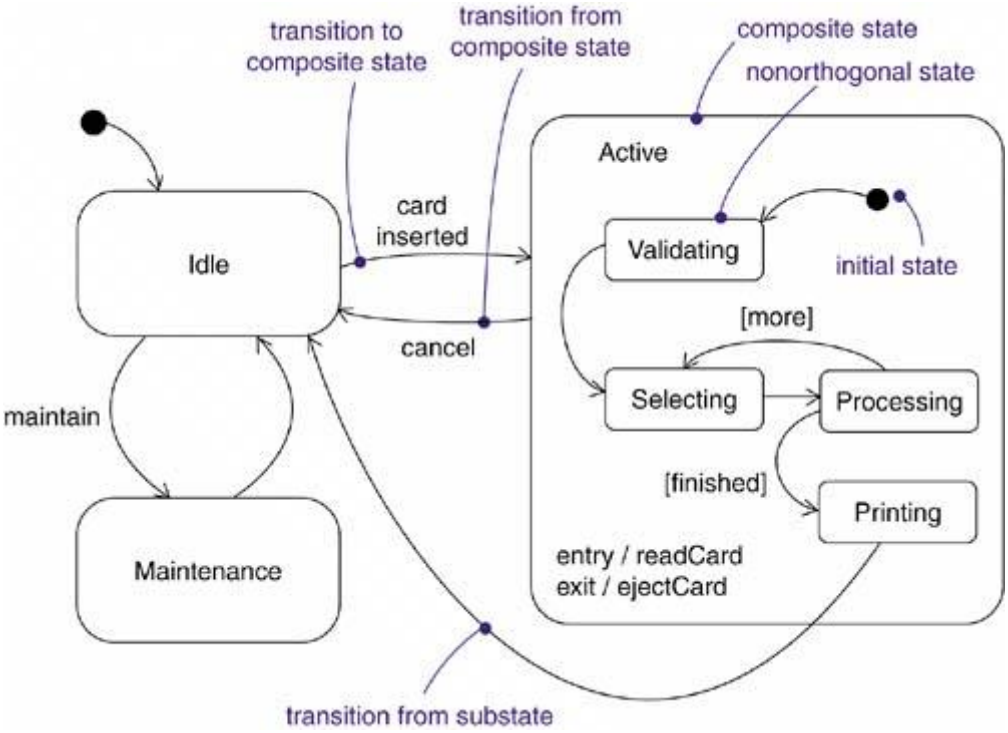
Nonorthogonal Substates

Consider the problem of modeling the behavior of an ATM. This system might be in one of three basic states: `Idle` (waiting for customer interaction), `Active` (handling a customer's transaction), and `Maintenance` (perhaps having its cash store replenished). While `Active`, the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction, and then print a receipt. After printing, the ATM returns to the `Idle` state. You might represent these stages of behavior as the states `Validating`, `Selecting`, `Processing`, and `Printing`. It would even be desirable to let the customer select and process multiple transactions after `Validating` the account and before `Printing` a final receipt.

The problem here is that, at any stage in this behavior, the customer might decide to cancel the transaction, returning the ATM to its `Idle` state. Using flat state machines, you can achieve that effect, but it's quite messy. Because the customer might cancel the transaction at any point, you'd have to include a suitable transition from every state in the `Active` sequence. That's messy because it's easy to forget to include these transitions in all the right places, and many such interrupting events means you end up with a multitude of transitions zeroing in on the same target state from various sources, but with the same event trigger, guard condition, and action.

Using nested substates, there's a simpler way to model this problem, as Figure 22-5 shows. Here, the `Active` state has a substructure, containing the substates `Validating`, `Selecting`, `Processing`, and `Printing`. The state of the ATM changes from `Idle` to `Active` when the customer enters a credit card in the machine. On entering the `Active` state, the entry action `readCard` is performed. Starting with the initial state of the substructure, control passes to the `Validating` state, then to the `Selecting` state, and then to the `Processing` state. After `Processing`, control may return to `Selecting` (if the customer has selected another transaction) or it may move on to `Printing`. After `Printing`, there's a completion transition back to the `Idle` state. Notice that the `Active` state has an exit action, which ejects the customer's credit card.

Figure 22-5. Sequential Substates



Notice also the transition from the `Active` state to the `Idle` state, triggered by the event `cancel`. In any substate of `Active`, the customer might cancel the transaction, and that returns the ATM to the `Idle` state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the `Active` state, no matter what caused a transition out of that state). Without substates, you'd need a transition triggered by `cancel` on every substructure state.

Substates such as `Validating` and `Processing` are called nonorthogonal, or disjoint, substates. Given a set of nonorthogonal substates in the context of an enclosing composite state, the object is said to be in the composite state and in only one of those substates (or the final state) at a time. Therefore, nonorthogonal substates partition the state space of the composite state into disjoint states.

From a source outside an enclosing composite state, a transition may target the composite state or it may target a substate. If its target is the composite state, the nested state machine must include an initial state, to which control passes after entering the composite state and after performing its entry action, if any. If its target is the nested state, control passes to the nested state after performing the entry action (if any) of the composite state and then the entry action (if any) of the substate.

A transition leading out of a composite state may have as its source the composite state or a substate. In either case, control first leaves the nested state (and its exit action, if any, is executed), then it leaves the composite state (and its exit action, if any, is executed). A transition whose source is the composite state essentially cuts short (interrupts) the activity of the nested state machine. The completion transition of a composite state is taken when control reaches the final substate within the composite state.

Note

A nested nonorthogonal state machine may have at most one initial substate and one final substate.

History States

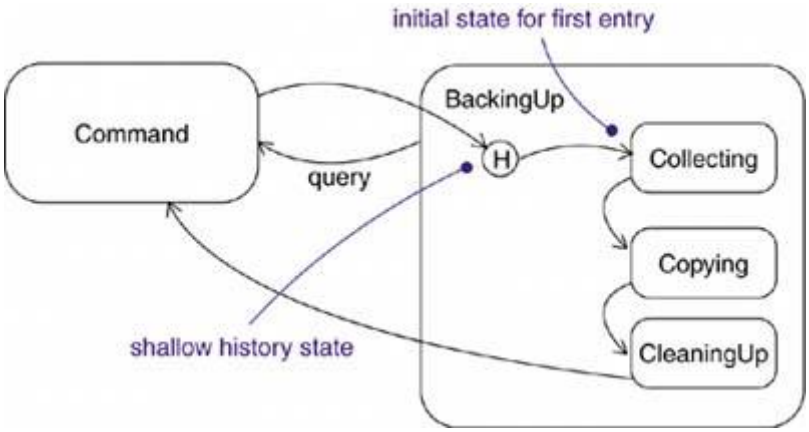
A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.

Unless otherwise specified, when a transition enters a composite state, the action of the nested state machine starts over again at its initial state (unless, of course, the transition targets a substate directly). However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. For example, in modeling the behavior of an agent that does an unattended backup of computers across a network, you'd like it to remember where it was in the process if it ever gets interrupted by, for example, a query from the operator.

Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. In this way, leaving the composite state would cause the last substate to be remembered; entering the composite state would transition to the proper substate. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.

In the UML, a simpler way to model this idiom is by using history states. A history state allows a composite state that contains nonorthogonal substates to remember the last substate that was active in it prior to the transition from the composite state. As Figure 22-6 shows, you represent a shallow history state as a small circle containing the symbol `H`.

Figure 22-6. History State



If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history. This is the meaning of the single transition from the history state to a sequential substate such as `Collecting`. The target of this transition specifies the initial state of the nested state machine the first time it is entered. Continuing, suppose that while in the `BackingUp` state and the `Copying` state, the `query` event is posted. Control leaves `Copying` and `BackingUp` (dispatching their exit actions as necessary) and returns to the `Command` state. When the action of `Command` completes, the completion transition returns to the history state of the composite state `BackingUp`. This time, because there is a history to the nested state machine, control passes back to the `Copying` statethus bypassing the `Collecting` statebecause `Copying` was the last substate active prior to the transition from the state `BackingUp`.

Note

The symbol `H` designates a shallow history, which remembers only the history of the immediate nested state machine. You can also specify deep history, shown as a small circle containing the symbol `H*`. Deep history remembers down to the innermost nested state at any depth. If you have only one level of nesting, shallow and deep history states are semantically equivalent. If you have more than one level of nesting, shallow history remembers only the outermost nested state; deep history remembers the innermost nested state at any depth.

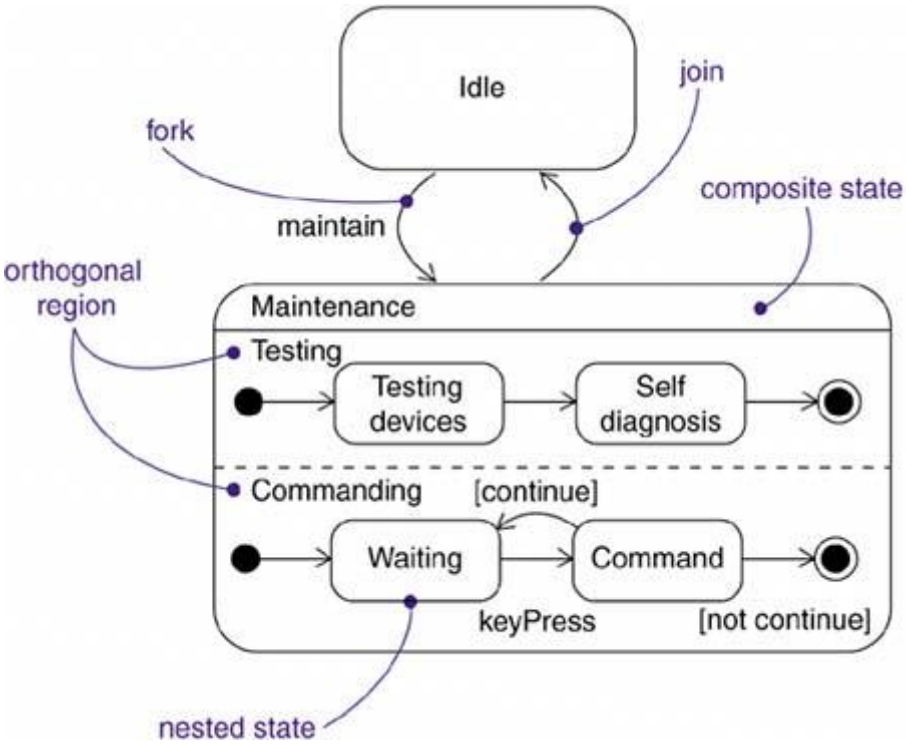
In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

Orthogonal Substates

Nonorthogonal substates are the most common kind of nested state machine you'll encounter. In certain modeling situations, however, you'll want to specify orthogonal regions. These regions let you specify two or more state machines that execute in parallel in the context of the enclosing object.

For example, Figure 22-7 shows an expansion of the `Maintenance` state from Figure 21-5. `Maintenance` is decomposed into two orthogonal regions, `Testing` and `Commanding`, shown by nesting them in the `Maintenance` state but separating them from one another with a dashed line. Each of these orthogonal regions is further decomposed into substates. When control passes from the `Idle` to the `Maintenance` state, control then forks to two concurrent flowsthe enclosing object will be in both the `Testing` region and the `Commanding` region. Furthermore, while in the `Commanding` region, the enclosing object will be in the `Waiting` or the `Command` state.

Figure 22-7. Concurrent Substates



Note

This is what distinguishes nonorthogonal substates and orthogonal substates. Given two or more nonorthogonal substates at the same level, an object will be in one of those substates or the other. Given two or more orthogonal regions at the same level, an object will be in a state from each of the orthogonal regions.

Execution of these two orthogonal regions continues in parallel. Eventually, each nested state machine reaches its final state. If one orthogonal region reaches its final state before the other, control in that region waits at its final state. When both nested state machines reach their final states, control from the two orthogonal regions joins back into one flow.

Whenever there's a transition to a composite state decomposed into orthogonal regions, control forks into as many concurrent flows as there are orthogonal regions. Similarly, whenever there's a transition from a composite substate decomposed into orthogonal regions, control joins back into one flow. This holds true in all cases. If all orthogonal regions reach their final states, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow.

Note

Each orthogonal region may have an initial, final, and history state.

Fork and Join

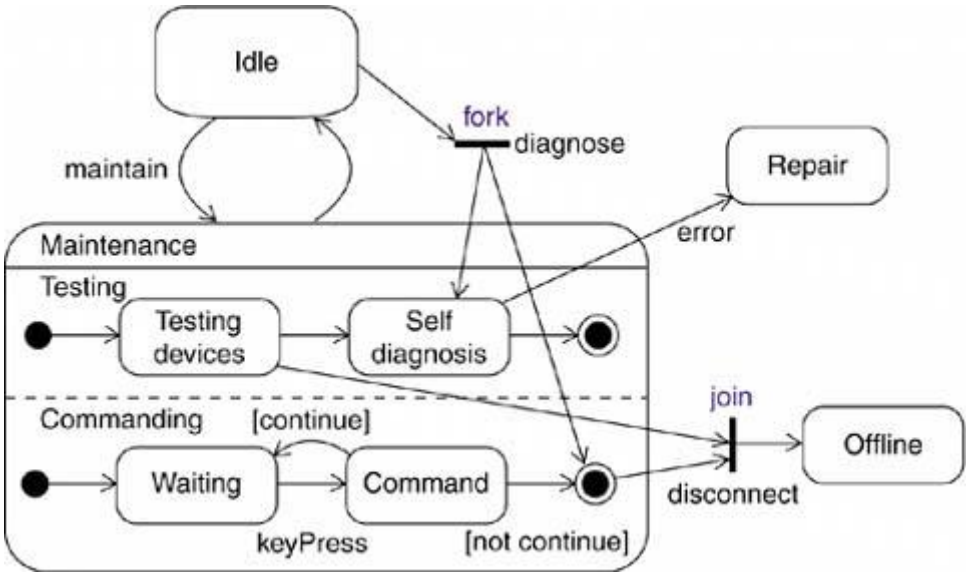
Usually, entry to a composite state with orthogonal regions goes to the initial state of each orthogonal region. It is also possible to transition from an external state directly to one or more orthogonal states. This is called a fork, because control passes from a single state to several orthogonal states. It is shown as a heavy black line with one incoming arrow and several outgoing arrows, each to one of the orthogonal states. There must be at most one target state in each orthogonal region. If one or more orthogonal regions have no target states, then the initial state of those regions is implicitly chosen. A transition to a single orthogonal state within a composite state is also an implicit fork; the initial states of all the other orthogonal regions are implicitly part of the fork.

Similarly, a transition from any state within a composite state with orthogonal regions forces an exit from all the orthogonal regions. Such a transition often represents an error condition that forces termination of parallel computations.

A join is a transition with two or more incoming arrows and one outgoing arrow. Each incoming arrow must come from a state in a different orthogonal region of the same composite state. The join may have a trigger event. The join transition is effective only if all of the source states are active; the status of other orthogonal regions in the composite state is irrelevant. If the event occurs, control leaves all of the orthogonal regions in the composite state, not just the ones with arrows from them.

Figure 22-8 shows a variation on the previous example with explicit fork and join transitions. The transition `maintain` to the composite state `Maintenance` is still an implicit fork into the default initial states of the orthogonal regions. In this example, however, there is also an explicit fork from `Idle` into the two nested states `Self diagnose` and the final state of the `Commanding` region. (A final state is a real state and can be the target of a transition.) If an error event occurs while the `Self diagnose` state is active, the implicit join transition to `Repair` fires: Both the `Self diagnose` state and whatever state is active in the `Commanding` region are exited. There is also an explicit join transition to the `Offline` state. This transition fires only if the `disconnect` event occurs while the `Testing devices` state and the final state of the `Commanding` region are both active; if both states are not active, the event has no effect.

Figure 22-8. Fork and join transitions



Active Objects

Another way to model concurrency is by using active objects. Thus, rather than partitioning one object's state machine into two (or more) concurrent regions, you could define two active objects, each of which is responsible for the behavior of one of the concurrent regions. If the behavior of one of these concurrent flows is affected by the state of the other, you'll want to model this using orthogonal regions. If the behavior of one of these concurrent flows is affected by messages sent to and from the other, you'll want to model this using active objects. If there's little or no communication between the concurrent flows, then the approach you choose is a matter of taste, although most of the time, using active objects makes your design decisions more obvious.

Common Modeling Techniques

Modeling the Lifetime of an Object

The most common purpose for which you'll use state machines is to model the lifetime of an object, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state machine models the behavior of a single object over its lifetime, such as you'll find with user interfaces, controllers, and devices.

When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior. Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction.

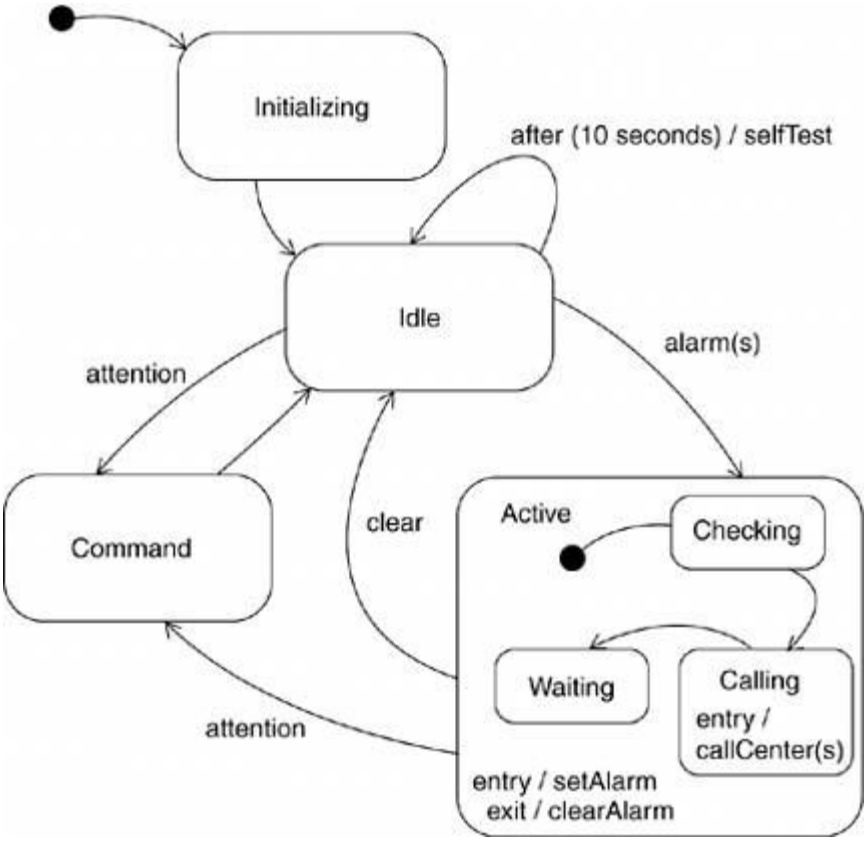
To model the lifetime of an object,

- Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
    - If the context is a class or a use case, find the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.

    - If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
- Expand these states as necessary by using substates.
- Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure 22-9 shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house.

Figure 22-9. Modeling the Lifetime of An Object



In the lifetime of this controller class, there are four main states: `Initializing` (the controller is starting up), `Idle` (the controller is ready and waiting for alarms or commands from the user), `Command` (the controller is processing commands from the user), and `Active` (the controller is processing an alarm condition). When the controller object is first created, it moves first to the `Initializing` state and then unconditionally to the `Idle` state. The details of these two states are not shown, other than the self-transition with the time event in the `Idle` state. This kind of time event is commonly found in embedded systems, which often have a heartbeat timer that causes a periodic check of the system's health.

Control passes from the `Idle` state to the `Active` state on receipt of an `alarm` event (which includes the parameter `s`, identifying the sensor that was tripped). On entering the `Active` state, `setAlarm` is performed as the entry action, and control then passes first to the `Checking` state (validating the alarm), then to the `Calling` state (calling the alarm company to register the alarm), and finally to the `Waiting` state.
The `Active` and `Waiting` states are exited only upon `clear`ing the alarm or by the user signaling the controller for `attention`, presumably to issue a command.

Notice that there is no final state. That, too, is common in embedded systems, which are intended to run indefinitely.