# 5

# State Modeling

You can best understand a system by first examining its static structure—that is, the structure of its objects and their relationships to each other at a single moment in time (the class model). Then you should examine changes to the objects and their relationships over time (the state model). The state model describes the sequences of operations that occur in response to external stimuli, as opposed to what the operations do, what they operate on, or how they are implemented.

The state model consists of multiple state diagrams, one for each class with temporal behavior that is important to an application. The state diagram is a standard computer science concept (a graphical representation of finite state machines) that relates events and states. Events represent external stimuli and states represent values of objects. You should master the material in this chapter before proceeding in the book.

## 5.1 Events

An ***event*** is an occurrence at a point in time, such as *user depresses left button* or *flight 123 departs from Chicago*. Events often correspond to verbs in the past tense (*power turned on, alarm set*) or to the onset of some condition (*paper tray becomes empty, temperature becomes lower than freezing*). By definition, an event happens instantaneously with regard to the time scale of an application. Of course, nothing is really instantaneous; an event is simply an occurrence that an application considers atomic and fleeting. The time at which an event occurs is an implicit attribute of the event. Temporal phenomena that occur over an interval of time are properly modeled with a state.

One event may logically precede or follow another, or the two events may be unrelated. Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are causally related. Flight 123 may depart before or after flight 456 departs Rome; the two events are causally unrelated. Two events that are causally unrelated are said to be ***concurrent;*** they

have no effect on each other. If the communications delay between two locations exceeds the difference in event times, then the events must be concurrent because they cannot influence each other. Even if the physical locations of two events are not distant, we consider the events concurrent if they do not affect each other. In modeling a system we do not try to establish an ordering between concurrent events because they can occur in any order.

Events include error conditions as well as normal occurrences. For example, *motor jammed, transaction aborted,* and *timeout* are typical error events. There is nothing different about an error event; only our interpretation makes it an "error."

The term *event* is often used ambiguously. Sometimes it refers to an instance, at other times to a class. In practice, this ambiguity is usually not a problem and the precise meaning is apparent from the context. If necessary, you can say *event occurrence* or *event type* to be precise.

There are several kinds of events. The most common are the signal event, the change event, and the time event.

### 5.1.1 Signal Event

A ***signal*** is an explicit one-way transmission of information from one object to another. It is different from a subroutine call that returns a value. An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.

A ***signal event*** is the event of sending or receiving a signal. Usually we are more concerned about the receipt of a signal, because it causes effects in the receiving object. Note the difference between *signal* and *signal event*—a signal is a message between objects while a signal event is an occurrence in time.

Every signal transmission is a unique occurrence, but we group them into *signal classes* and give each signal class a name to indicate common structure and behavior. For example, *UA flight 123 departs from Chicago on January 10, 1991* is an instance of signal class *FlightDeparture*. Some signals are simple occurrences, but most signal classes have attributes indicating the values they convey. For example, as Figure 5.1 shows, *FlightDeparture* has attributes *airline, flightNumber, city,* and *date*. The UML notation is the keyword *signal* in guillemets («») above the signal class name in the top section of a box. The second section lists the signal attributes.

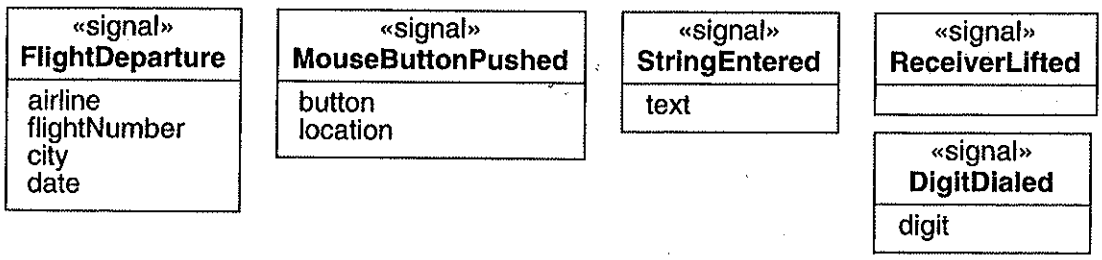


**Figure 5.1 Signal classes and attributes.** A signal is an explicit one-way transmission of information from one object to another.

### 5.1.2 Change Event

A ***change event*** is an event that is caused by the satisfaction of a boolean expression. The intent of a change event is that the expression is continually tested—whenever the expression changes from false to true, the event happens. Of course, an implementation would not *continuously* check a change event, but it must check often enough so that it seems continuous from an application perspective.

The UML notation for a change event is the keyword *when* followed by a parenthesized boolean expression. Figure 5.2 shows several examples of change events.

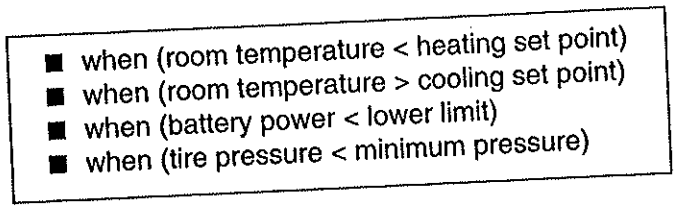


**Figure 5.2 Change events.** A change event is an event that is caused by the satisfaction of a boolean expression.

### 5.1.3 Time Event

A ***time event*** is an event caused by the occurrence of an absolute time or the elapse of a time interval. As Figure 5.3 shows, the UML notation for an absolute time is the keyword *when* followed by a parenthesized expression involving time. The notation for a time interval is the keyword *after* followed by a parenthesized expression that evaluates to a time duration.

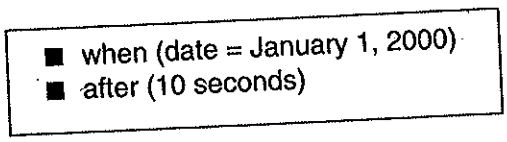


**Figure 5.3 Time events.** A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.

## 5.2 States

A ***state*** is an abstraction of the values and links of an object. Sets of values and links are grouped together into a state according to the gross behavior of objects. For example, the state of a bank is either solvent or insolvent, depending on whether its assets exceed its liabilities. States often correspond to verbs with a suffix of "ing" (*Waiting*, *Dialing*) or the duration of some condition (*Powered*, *BelowFreezing*).

Figure 5.4 shows the UML notation for a state—a rounded box containing an optional state name. Our convention is to list the state name in boldface, center the name near the top of the box, and capitalize the first letter.


**Figure 5.4 States.** A state is an abstraction of the values and links of an object.

In defining states, we ignore attributes that do not affect the behavior of the object, and lump together in a single state all combinations of values and links with the same response to events. Of course, every attribute has some effect on behavior or it would be meaningless, but often some attributes do not affect the sequence of control and you can regard them as simple parameter values within a state. Recall that the purpose of modeling is to focus on qualities that are relevant to the solution of an application problem and abstract away those that are irrelevant. The three UML models (class, state, and interaction) present different views of a system for which the particular choice of attributes and values are not equally important. For example, except for leading 0s and 1s, the exact digits dialed do not affect the control of the phone line, so we can summarize them all with state *Dialing* and track the phone number as a parameter. Sometimes, all possible values of an attribute are important, but usually only when the number of possible values is small.

The objects in a class have a finite number of possible states—one or possibly some larger number. Each object can only be in one state at a time. Objects may parade through one or more states during their lifetime. At a given moment of time, the various objects for a class can exist in a multitude of states.

A state specifies the response of an object to input events. All events are ignored in a state, except those for which behavior is explicitly prescribed. The response may include the invocation of behavior or a change of state. For example, if a digit is dialed in state *Dial tone*, the phone line drops the dial tone and enters state *Dialing*; if the receiver is replaced in state *Dial tone*, the phone line goes dead and enters state *Idle*.

There is a certain symmetry between events and states as Figure 5.5 illustrates. Events represent points in time; states represent intervals of time. A state corresponds to the interval between two events received by an object. For example, after the receiver is lifted and before the first digit is dialed, the phone line is in state *Dial tone*. The state of an object depends on past events, which in most cases are eventually hidden by subsequent events. For example, events that happened before the phone is hung up do not affect future behavior; the *Idle* state "forgets" events received prior to the receipt of the *hang up* signal.

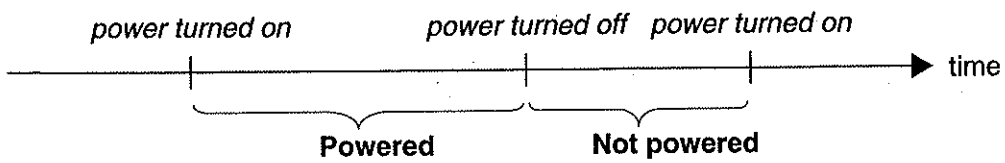


**Figure 5.5 Event vs. state.** Events represent points in time; states represent intervals of time.

Both events and states depend on the level of abstraction. For example, a travel agent planning an itinerary would treat each segment of a journey as a single event; a flight status

board in an airport would distinguish departures and arrivals; an air traffic control system would break each flight into many geographical legs.

You can characterize a state in various ways, as Figure 5.6 shows for the state *Alarm ringing* on a watch. The state has a suggestive name and a natural-language description of its purpose. The event sequence that leads to the state consists of setting the alarm, doing anything that doesn't clear the alarm, and then having the target time occur. A declarative condition for the state is given in terms of parameters, such as *current* and *target time*; the alarm stops ringing after 20 seconds. Finally, a stimulus-response table shows the effect of events *current time* and *button pushed*, including the response that occurs and the next state. The different descriptions of a state may overlap.

**State**: *AlarmRinging*

**Description**: alarm on watch is ringing to indicate target time

**Event sequence that produces the state:**

*setAlarm* (*targetTime*)
any sequence not including *clearAlarm*
when (*currentTime* = *targetTime*)

**Condition that characterizes the state:**

alarm = on, alarm set to *targetTime*, *targetTime* ≤ *currentTime* ≤ *targetTime* + 20 seconds, and no button has been pushed since *targetTime*

**Events accepted in the state:**

| event | response | next state |
|---|---|---|
| when (*currentTime* = *targetTime* + 20) | *resetAlarm* | *normal* |
| *buttonPushed* (any button) | *resetAlarm* | *normal* |

**Figure 5.6 Various characterizations of a state.** A state specifies the response of an object to input events.

Can links have state? In as much as they can be considered objects, links can have state. As a practical matter, it is generally sufficient to associate state only with objects.

## 5.3 Transitions and Conditions

A ***transition*** is an instantaneous change from one state to another. For example, when a called phone is answered, the phone line transitions from the *Ringing* state to the *Connected* state. The transition is said to ***fire*** upon the change from the source state to the target state. The origin and target of a transition usually are different states, but may be the same. A transition fires when its event occurs (unless an optional guard condition causes the event to be ignored). The choice of next state depends on both the original state and the event received.

An event may cause multiple objects to transition; from a conceptual point of view such transitions occur concurrently.

A ***guard condition*** is a boolean expression that must be true in order for a transition to occur. For example, a traffic light at an intersection may change only if a road has cars waiting. A guarded transition fires when its event occurs, but only if the guard condition is true. For example, "when you go out in the morning (event), if the temperature is below freezing (condition), then put on your gloves (next state)." A guard condition is checked only once, at the time the event occurs, and the transition fires if the condition is true. If the condition becomes true later, the transition does not then fire. Note that a guard condition is different from a change event—a guard condition is checked only once while a change event is, in effect, checked continuously.

Figure 5.7 shows guarded transitions for traffic lights at an intersection. One pair of electric eyes checks the north-south left turn lanes; another pair checks the east-west turn lanes. If no car is in the north-south and/or east-west turn lanes, then the traffic light control logic is smart enough to skip the left turn portion of the cycle.

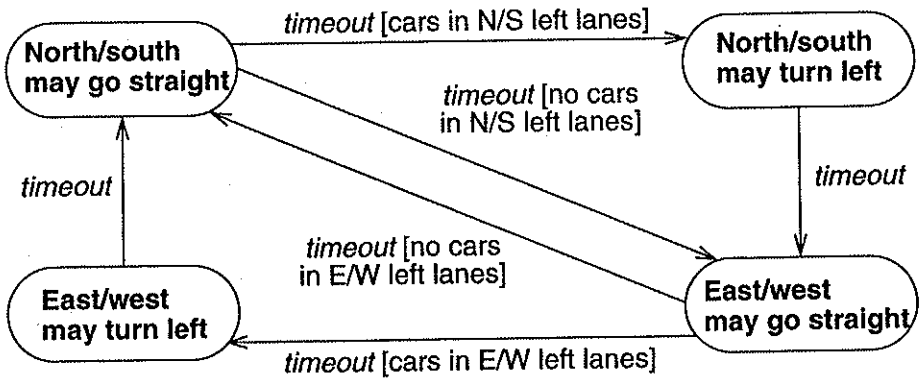


**Figure 5.7 Guarded transitions.** A transition is an instantaneous change from one state to another. A guard condition is a boolean expression that must be true in order for a transition to occur.

The UML notation for a transition is a line from the origin state to the target state. An arrowhead points to the target state. The line may consist of several line segments. An event may label the transition and be followed by an optional guard condition in square brackets. By convention, we usually confine line segments to a rectilinear grid. We italicize the event name and show the condition in normal font.

## 5.4 State Diagrams

A ***state diagram*** is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the state sequences caused by event sequences. State names must be unique within the scope of a state diagram. All objects in a class execute the state diagram for that class, which models their common behavior. You can implement

state diagrams by direct interpretation or by converting the semantics into equivalent programming code.

The ***state model*** consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions. The individual state diagrams interact by passing events and through the side effects of guard conditions. Some events and guard conditions appear in a single state diagram; others appear in multiple state diagrams for the purpose of coordination. This chapter covers only individual state diagrams; Chapter 6 discusses state models of interacting diagrams.

A class with more than one state has important temporal behavior. Similarly, a class is temporally important if it has a single state with multiple responses to events. You can represent state diagrams with a single state in a simple nongraphical form—a stimulus–response table listing events and guard conditions and the ensuing behavior.

### 5.4.1 Sample State Diagram

Figure 5.8 shows a state diagram for a telephone line. The diagram concerns a phone line and not the caller nor callee. The diagram contains sequences associated with normal calls as well as some abnormal sequences, such as timing out while dialing or getting busy lines. The UML notation for a state diagram is a rectangle with its name in a small pentagonal tag in the upper left corner. The constituent states and transitions lie within the rectangle.

At the start of a call, the telephone line is idle. When the phone is removed from the hook, it emits a dial tone and can accept the dialing of digits. Upon entry of a valid number, the phone system tries to connect the call and route it to the proper destination. The connection can fail if the number or trunk are busy. If the connection is successful, the called phone begins ringing. If the called party answers the phone, a conversation can occur. When the called party hangs up, the phone disconnects and reverts to idle when put on hook again.

Note that the receipt of the signal *onHook* causes a transition from any state to *Idle* (the bundle of transitions leading to *Idle*). Chapter 6 will show a more general notation that represents events applicable to groups of states with a single transition.

States do not totally define all values of an object. For example, state *Dialing* includes all sequences of incomplete phone numbers. It is not necessary to distinguish between different numbers as separate states, since they all have the same behavior, but the actual number dialed must of course be saved as an attribute.

If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire. If an event occurs and no transition matches it, then the event is ignored. If more than one transition matches an event, only one transition will fire, but the choice is nondeterministic.

### 5.4.2 One-shot State Diagrams

State diagrams can represent continuous loops or one-shot life cycles. The diagram for the phone line is a continuous loop. In describing ordinary usage of the phone, we do not know or care how the loop is started. (If we were describing installation of new lines, the initial state would be important.)

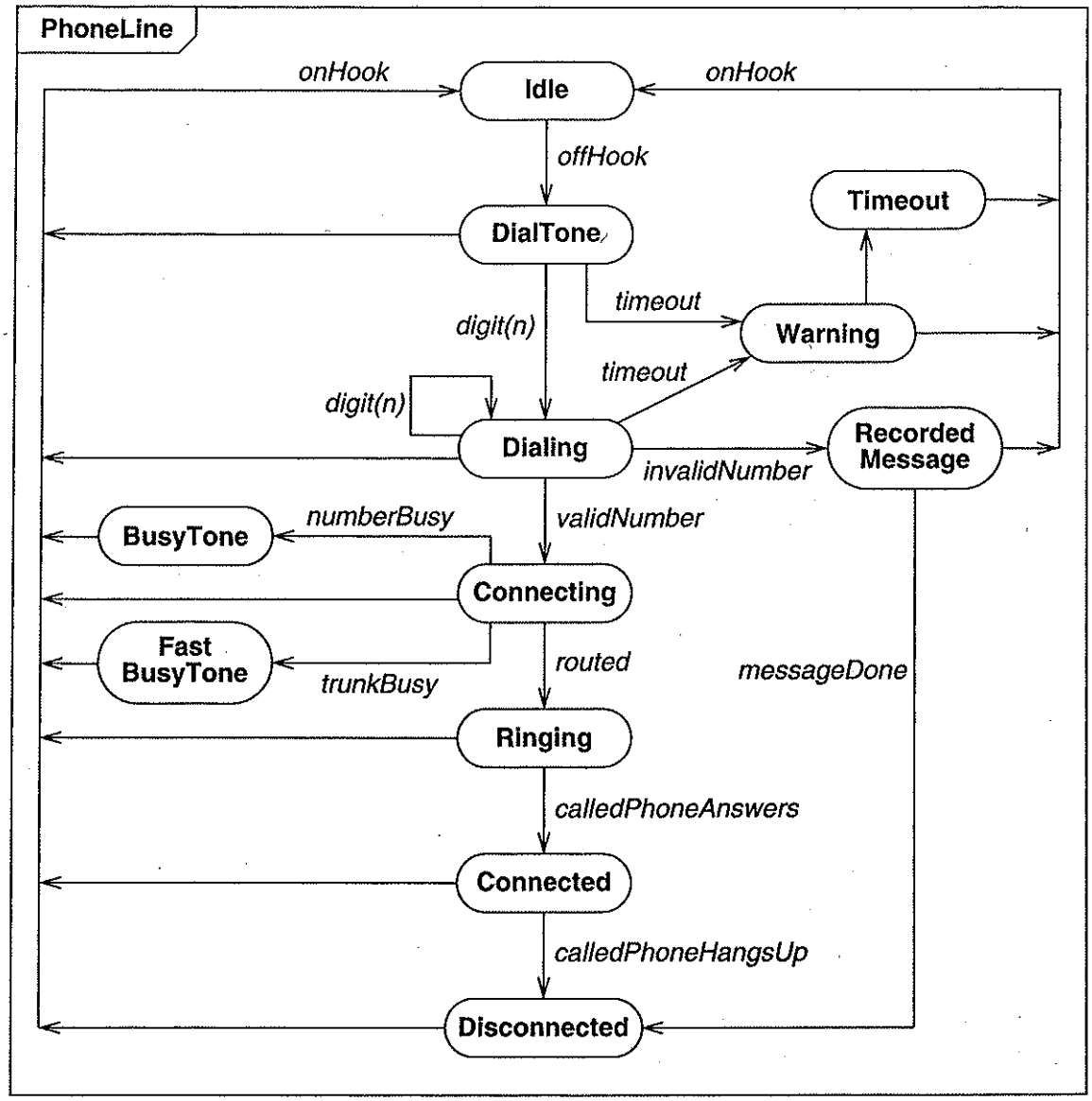


**Figure 5.8 State diagram for a telephone line.** A state diagram specifies the state sequences caused by event sequences.

One-shot state diagrams represent objects with finite lives and have initial and final states. The initial state is entered on creation of an object; entry of the final state implies destruction of the object. Figure 5.9 shows a simplified life cycle of a chess game with a default initial state (solid circle) and a default final state (bull's eye).

As an alternate notation, you can indicate initial and final states via entry and exit points. In Figure 5.10 the *start* entry point leads to white's first turn, and the chess game eventually ends with one of three possible outcomes. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.

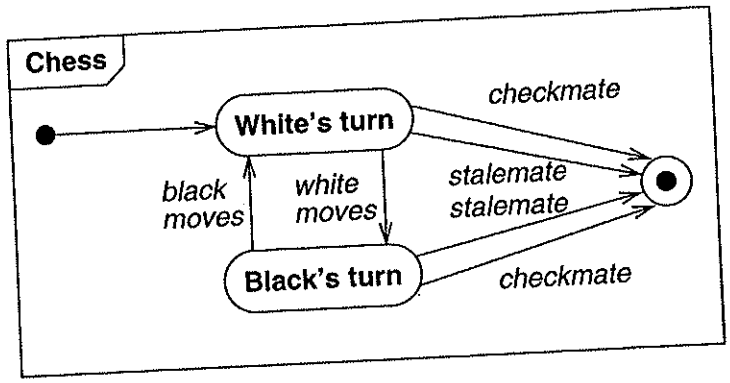


**Figure 5.9 State diagram for chess game.** One-shot diagrams represent objects with finite lives.

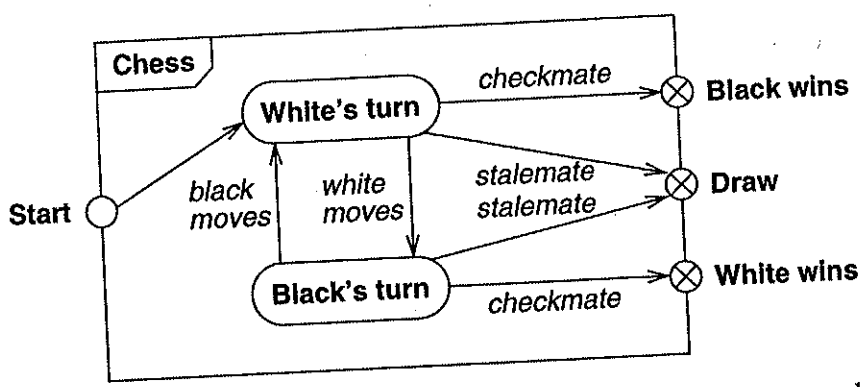


**Figure 5.10 State diagram for chess game.** You can also show one-shot diagrams by using entry and exit points.

### 5.4.3 *Summary of Basic State Diagram Notation*

Figure 5.11 summarizes the basic UML syntax for state diagrams.

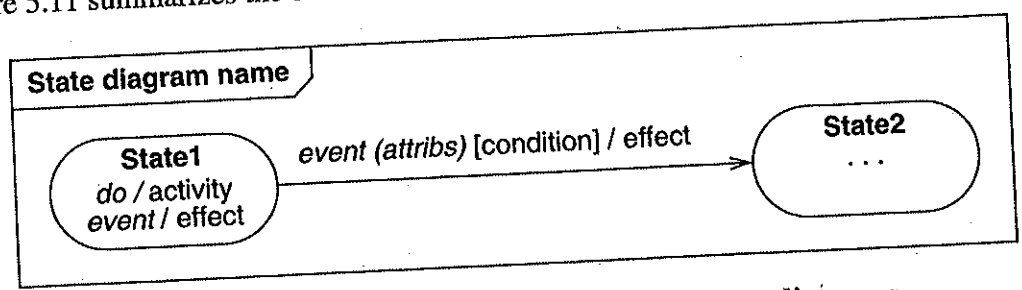


**Figure 5.11 Summary of basic notation for state diagrams.**

- **State.** Drawn as a rounded box containing an optional name. A special notation is avai able for initial states (a solid circle) and final states (a bull's-eye or encircled "x").

- **Transition.** Drawn as a line from the origin state to the target state. An arrowhead points to the target state. The line may consist of several line segments.
- **Event.** A signal event is shown as a label on a transition and may be followed by parenthesized attributes. A change event is shown with the keyword *when* followed by a parenthesized boolean expression. A time event is shown with the keyword *when* followed by a parenthesized expression involving time or the keyword *after* followed by a parenthesized expression that evaluates to a time duration.
- **State diagram.** Enclosed in a rectangular frame with the diagram name in a small pentagonal tag in the upper left corner.
- **Guard condition.** Optionally listed in square brackets after an event.
- **Effects** (to be explained in next section). Can be attached to a transition or state and are listed after a slash ("/"). Multiple effects are separated with a comma and are performed concurrently. (You can create intervening states if you want multiple effects to be performed in sequence.)

We also recommend some style conventions. We list the state name in boldface with the first letter capitalized. We italicize event names with the initial letter in lower case. Guard conditions and effects are in normal font and also have the initial letter in lower case. We try to confine transition line segments to a rectilinear grid.

## 5.5 State Diagram Behavior

State diagrams would be of little use if they just described events. A full description of an object must specify what the object does in response to events.

### 5.5.1 Activity Effects

An ***effect*** is a reference to a behavior that is executed in response to an event. An ***activity*** is the actual behavior that can be invoked by any number of effects. For example, *disconnectPhoneLine* might be an activity that is executed in response to an *onHook* event for Figure 5.8. An activity may be performed upon a transition, upon the entry to or exit from a state, or upon some other event within a state.

Activities can also represent internal control operations, such as setting attributes or generating other events. Such activities have no real-world counterparts but instead are mechanisms for structuring control within an implementation. For example, a program might increment an internal counter every time a particular event occurs.

The notation for an activity is a slash ("/") and the name (or description) of the activity, following the event that causes it. The keyword *do* is reserved for indicating an ongoing activity (to be explained) and may not be used as an event name. Figure 5.12 shows the state diagram for a pop-up menu on a workstation.When the right button is depressed, the menu is displayed; when the right button is released, the menu is erased. While the menu is visible, the highlighted menu item is updated whenever the cursor moves.

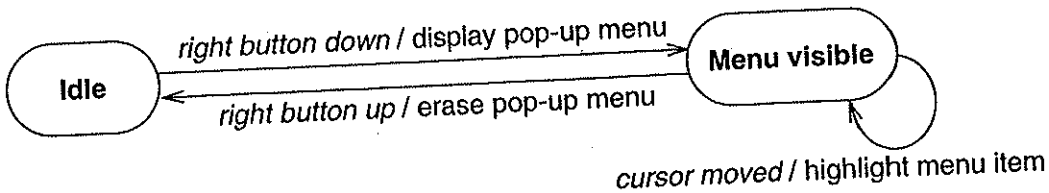


**Figure 5.12 Activities for pop-up menu.** An activity is behavior that can be executed in response to an event.

### 5.5.2 Do-Activities

A ***do-activity*** is an activity that continues for an extended time. By definition, a do-activity can only occur within a state and cannot be attached to a transition. For example, the warning light may flash during the *Paper jam* state for a copy machine (Figure 5.13). Do-activities include continuous operations, such as displaying a picture on a television screen, as well as sequential operations that terminate by themselves after an interval of time, such as closing a valve.

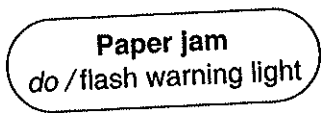


**Figure 5.13 Do-activity for a copy machine.** A do-activity is an activity that continues for an extended time.

The notation "*do /*" denotes a do-activity that may be performed for all or part of the duration that an object is in a state. A do-activity may be interrupted by an event that is received during its execution; such an event may or may not cause a transition out of the state containing the do-activity. For example, a robot moving a part may encounter resistance, causing it to cease moving.

### 5.5.3 Entry and Exit Activities

As an alternative to showing activities on transitions, you can bind activities to entry or to exit from a state. There is no difference in expressive power between the two notations, but frequently all transitions into a state perform the same activity, in which case it is more concise to attach the activity to the state.

For example, Figure 5.14 shows the control of a garage door opener. The user generates *depress* events with a pushbutton to open and close the door. Each event reverses the direction of the door, but for safety the door must open fully before it can be closed. The control generates *motor up* and *motor down* activities for the motor. The motor generates *door open* and *door closed* events when the motion has been completed. Both transitions entering state *Opening* cause the door to open.

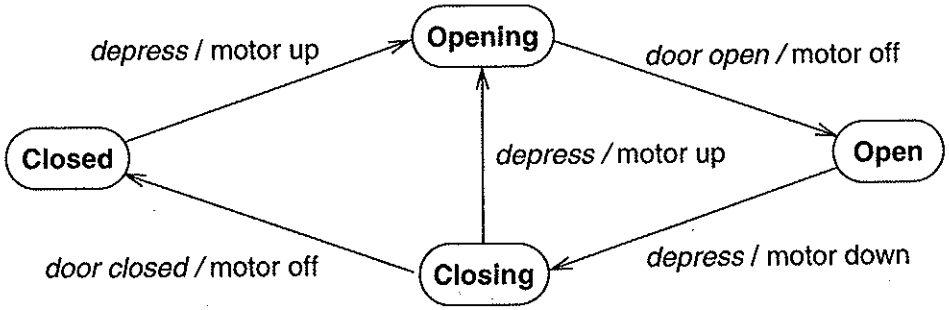


**Figure 5.14 Activities on transitions.** An activity may be bound to an event that causes a transition.

Figure 5.15 shows the same model using activities on entry to states. An entry activity is shown inside the state box following the keyword *entry* and a "/" character. Whenever the state is entered, by any incoming transition, the entry activity is performed. An entry activity is equivalent to attaching the activity to every incoming transition. If an incoming transition already has an activity, its activity is performed first.

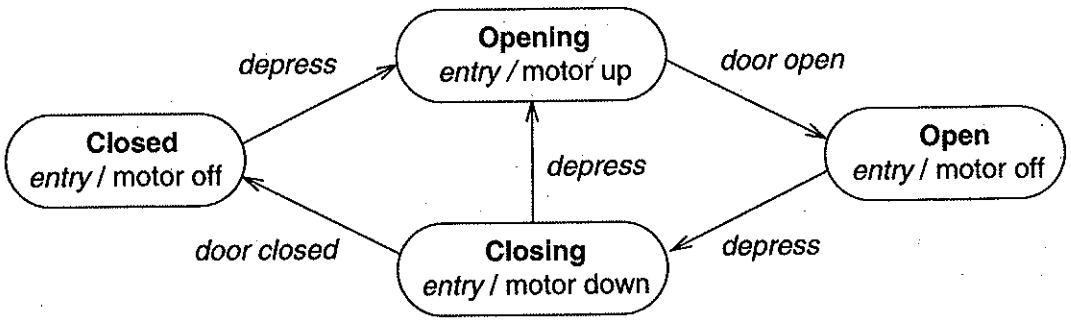


**Figure 5.15 Activities on entry to states.** An activity may also be bound to an event that occurs within a state.

Exit activities are less common than entry activities, but they are occasionally useful. An exit activity is shown inside the state box following the keyword *exit* and a "/" character. Whenever the state is exited, by any outgoing transition, the exit activity is performed first.

If a state has multiple activities, they are performed in the following order: activities on the incoming transition, entry activities, do-activities, exit activities, activities on the outgoing transition. Events that cause transitions out of the state can interrupt do-activities. If a do-activity is interrupted, the exit activity is still performed.

In general, any event can occur within a state and cause an activity to be performed. *Entry* and *exit* are only two examples of events that can occur. As Figure 5.16 shows, there is a difference between an event within a state and a self-transition; only the self-transition causes the entry and exit activities to be executed.

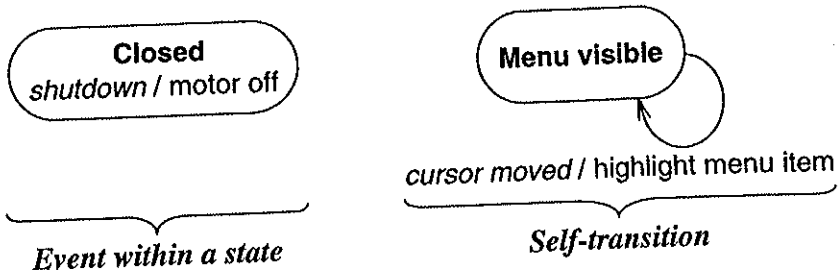


**Figure 5.16 Event within a state vs. self-transition.** A self-transition causes entry and exit activities to be executed. An event within a state does not.

### 5.5.4 Completion Transition

Often the sole purpose of a state is to perform a sequential activity. When the activity is completed, a transition to another state fires. An arrow without an event name indicates an automatic transition that fires when the activity associated with the source state is completed. Such unlabeled transitions are called ***completion transitions*** because they are triggered by the completion of activity in the source state.

A guard condition is tested only once, when the event occurs. If a state has one or more completion transitions, but none of the guard conditions are satisfied, then the state remains active and may become "stuck"—the completion event does not occur a second time, therefore no completion transition will fire later to change the state. If a state has completion transitions leaving it, normally the guard conditions should cover every possible outcome. You can use the special condition *else* to apply if all the other conditions are false. Do not use a guard condition on a completion transition to model waiting for a change of value. Instead model the waiting as a change event.

### 5.5.5 Sending Signals

An object can perform the activity of sending a signal to another object. A system of objects interacts by exchanging signals.

The activity "send *target.S(attributes)*" sends signal *S* with the given attributes to the target object or objects. For example, the phone line sends a *connect(phone number)* signal to the switcher when a complete phone number has been dialed. A signal can be directed at a set of objects or a single object. If the target is a set of objects, each of them receives a separate copy of the signal concurrently, and each of them independently processes the signal and determines whether to fire a transition. If the signal is always directed to the same object, the diagram can omit the target (but it must be supplied eventually in an implementation, of course).

If an object can receive signals from more than one object, the order in which concurrent signals are received may affect the final state; this is called a ***race condition***. For example, in Figure 5.15 the door may or may not remain open if the button is pressed at about the time the door becomes fully open. A race condition is not necessarily a design error, but concur-

rent systems frequently contain unwanted race conditions that must be avoided by careful design. A requirement of two signals being received simultaneously is never a meaningful condition in the real world, as slight variations in transmission speed are inherent in any distributed system.

### 5.5.6 Sample State Diagram with Activities

Figure 5.17 adds activities to the state diagram from Figure 5.8.

## 5.6 Practical Tips

The precise content of all models depends on application needs. The chapter has already mentioned the following practical tips, and we summarize them here for your convenience.

- **Abstracting values into states.** Consider only *relevant* attributes when defining a state. State diagrams need not use all attributes shown in a class model. (Section 5.2)
- **Parameters.** Parameterize events for incidental data that do not affect the flow of control. (Section 5.2)
- **Granularity of events and states.** Consider application needs when deciding on the granularity of events and states. (Section 5.2)
- **When to use state diagrams.** Construct state diagrams only for classes with meaningful temporal behavior. A class has important temporal behavior if it responds differently to various events or has more than one state. Not all classes require a state diagram. (Section 5.4)
- **Entry and exit activities.** When a state has multiple incoming transitions, and all transitions cause the same activity to occur, use an *entry* activity within the state rather than repeatedly listing the activity on transition arcs. Do likewise for *exit* activities. (Section 5.5.3)
- **Guard conditions.** Be careful with guard conditions so that an object does not become "stuck" in a state. (Section 5.5.4)
- **Race conditions.** Beware of unwanted race conditions in state diagrams. Race conditions may occur when a state can accept events from more than one object. (Section 5.5.5)

## 5.7 Chapter Summary

Event and state are the two elementary concepts in state modeling. An event is an occurrence at a point in time. A state is an abstraction of the values and links of an object. Events represent points in time; states represent intervals of time. An object may respond to certain events when it is in certain states. All events are ignored in a state, except those for which behavior is explicitly prescribed. The same event can have different effects (or no effect) in different states.

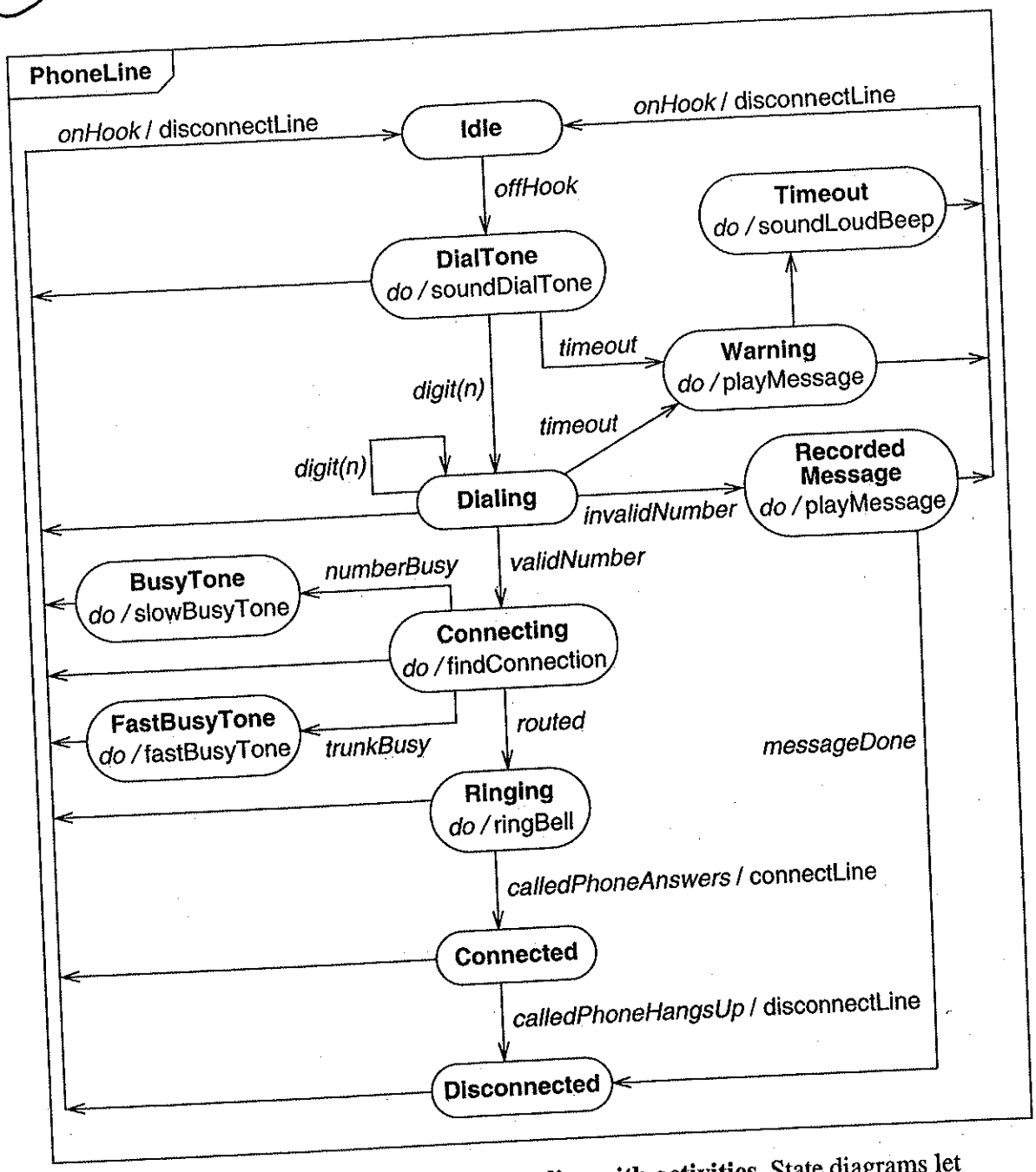


**Figure 5.17 State diagram for phone line with activities.** State diagrams let you express what objects do in response to events.

There are several kinds of events, such as a signal event, a change event, and a time event. A signal event is the sending or receipt of information communicated among objects. A change event is an event that is caused by the satisfaction of a boolean expression. A time event is an event caused by the occurrence of an absolute time or the elapse of a relative time.

A transition is an instantaneous change from one state to another and is caused by the occurrence of an event. An optional guard condition can cause the event to be ignored. A guard condition is a boolean expression that must be true in order for a transition to occur.

An effect is a reference to a behavior that is executed by objects in response to an event. An activity is the actual behavior that can be invoked by any number of effects. An activity may be performed upon a transition or upon an event within a state. A do-activity is an interruptible behavior that continues for an extended time. Consequently, a do-activity can occur only within a state and cannot be attached to a transition.

A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the possible states, what transitions are allowed between states, what events cause the transitions to occur, and what behavior is executed in response to events. A state diagram describes the common behavior for the objects in a class; as each object has its own values and links, so too each object has its own state or position in the state diagram. The state model consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions.

| | | | |
|---|---|---|---|
| activity | do-activity | race condition | state model |
| change event | effect | signal | time event |
| completion transition | event | signal event | transition |
| concurrency | fire (a transition) | state | |
| control | guard condition | state diagram | |

**Figure 5.18 Key concepts for Chapter 5**

## Bibliographic Notes

[Wieringa-98] has a thorough comparison of various ways for specifying software, including specification of the dynamic behavior of systems.

Finite state machines are a basic computer science concept and are described in any standard text on automata theory, such as [Hopcroft-01]. They are often described as recognizers or generators of formal languages. Basic finite state machines have limited expressive power. They have been extended with local variables and recursion as Augmented Transition Networks [Woods-70] and Recursive Transition Networks. These extensions expand the range of formal languages they can express but do little to address the combinatorial explosion that makes them unwieldy for practical control problems. (Chapter 6 addresses this.)

Traditional finite automata have been approached from a synchronous viewpoint. Petri nets [Reisig-92] formalize concurrency and synchronization of systems with distributed activity without resort to any notion of global time. Although they succeed well as an abstract conceptual model, they are too low-level and inexpressive to be useful for specifying large systems.

The need to specify interactive user interfaces has created several techniques for specifying control. This work is directed toward finding notations that clearly express powerful kinds of interactions while also being easily implementable. See [Green-86] for a comparison of some of these techniques.

The first edition of this book distinguished between *actions* (instantaneous behavior) and *activities* (lengthy behavior). UML2 has redefined both of these terms, and we have modified our explanation accordingly. UML2 now defines an activity as a specification of executable behavior and an action as a predefined primitive activity. In effect, the new definition of activity in UML2 subsumes the action and activity of the old book.

## References

[Green-86] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics 5,* 3 (July 1986), 244–275.

[Hopcroft-01] J.E. Hopcroft, Rejeev Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation., Second Edition,* Boston: Addison-Wesley, 2001.

[Reisig-92]. Wolfgang Reisig. *A Primer in Petri Net Design.* New York: Springer-Verlag, 1992.

[Wieringa-98] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys 30,* 4 (December 1998), 459–527.

[Woods-70] W.A. Woods. Transition network grammars for natural language analysis. *Communications of ACM 13,* 10 (October 1970), 591–606.

## Exercises

5.1 (6) An extension ladder has a rope, pulley, and latch for raising, lowering, and locking the extension. When the latch is locked, the extension is mechanically supported and you may safely climb the ladder. To release the latch, you raise the extension slightly with the rope. You may then freely raise or lower the extension. The latch produces a clacking sound as it passes over rungs of the ladder. The latch may be reengaged while raising the extension by reversing direction just as the latch is passing a rung. Prepare a state diagram of an extension ladder.

5.2 (4) A simple digital watch has a display and two buttons to set it, the A button and the B button. The watch has two modes of operation, display time and set time. In the display time mode, the watch displays hours and minutes, separated by a flashing colon.

The set time mode has two submodes, set hours and set minutes. The A button selects modes. Each time it is pressed, the mode advances in the sequence: display, set hours, set minutes, display, etc. Within the submodes, the B button advances the hours or minutes once each time it is pressed. Buttons must be released before they can generate another event. Prepare a state diagram of the watch.

5.3 (4) Figure E5.1 is a partially completed and simplified state diagram for the control of a telephone answering machine. The machine detects an incoming call on the first ring and answers the call with a prerecorded announcement. When the announcement is complete, the machine records the caller's message. When the caller hangs up, the machine hangs up and shuts off. Place the following in the diagram: call detected, answer call, play announcement, record message, caller hangs up, announcement complete.

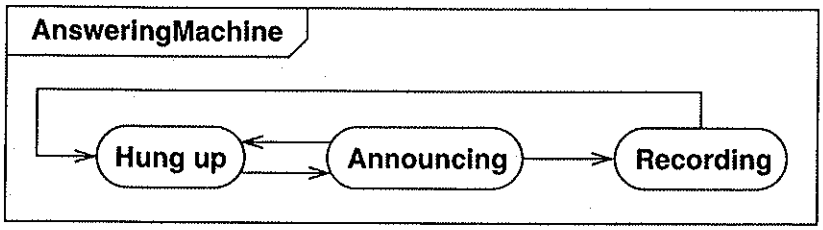


**Figure E5.1 Partial state diagram for an answering machine**

5.4 (7) The telephone answering machine in the previous exercise activates on the first ring. Revise the state diagram so that the machine answers after five rings. If someone answers the telephone before five rings, the machine should do nothing. Be careful to distinguish between five calls in which the telephone is answered on the first ring and one call that rings five times.

5.5 (3) In a personal computer, a disk controller is typically used to transfer a stream of bytes from a floppy disk drive to a memory buffer with the help of a host such as the central processing unit (CPU) or a direct memory access (DMA) controller. Figure E5.2 shows a partially completed and simplified state diagram for the control of the data transfer.

The controller signals the host each time a new byte is available. The data must then be read and stored before another byte is ready. When the disk controller senses the data has been read, it indicates that data is not available, in preparation for the next byte. If any byte is not read before the next one comes along, the disk controller asserts a data lost error signal until the disk controller is reset. Add the following to the diagram: reset, indicate data not available, indicate data available, data read by host, new data ready, indicate data lost.

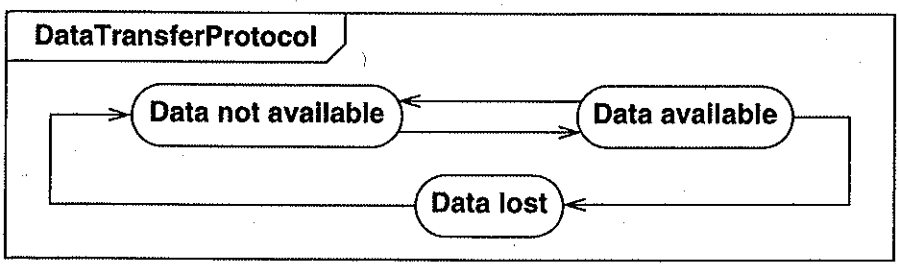


**Figure E5.2 Partially completed state diagram of a data transfer protocol**

5.6 (5) Figure E5.3 is a partially completed state diagram for one kind of motor control that is commonly used in household appliances. A separate appliance control determines when the motor should be on and continuously asserts *on* as an input to the motor control when the motor should be running.

When *on* is asserted, the motor control should start and run the motor. The motor starts by applying power to both the *start* and the *run* windings. A sensor, called a *starting relay*, determines when the motor has started, at which point the *start* winding is turned off, leaving only the *run* winding powered. Both windings are shut off when *on* is not asserted.

Appliance motors could be damaged by overheating if they are overloaded or fail to start. To protect against thermal damage, the motor control often includes an over-temperature sensor. If

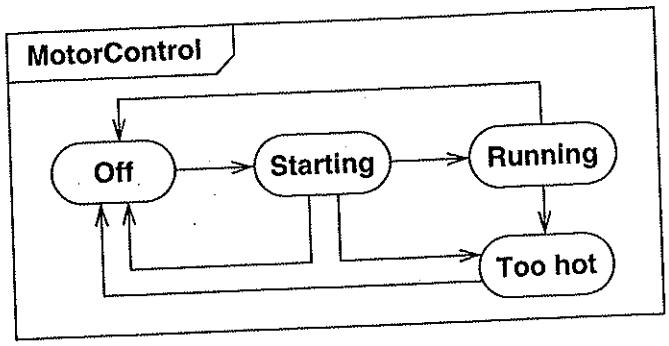


**Figure E5.3 Partially completed state diagram for a motor control**

the motor becomes too hot, the motor control removes power from both windings and ignores any *on* assertion until a reset button is pressed and the motor has cooled off.

Add the following to the diagram. Activities: apply power to run winding, apply power to start winding. Events: motor is overheated, on is asserted, on is no longer asserted, motor is running, reset. Condition: motor is not overheated.

5.7 (6) There was a single, continuously active input to the control in Exercise 5.6. In another common motor control, there are two pushbuttons, one for *start* and one for *stop*. To start the motor, the user presses the *start* button. The motor continues to run after the *start* button is released.To stop the motor, the user presses the *stop* button. The *stop* button takes precedence over the *start* button, so that the motor does not run while both buttons are pressed.

If both buttons are pressed and released, whether or not the motor starts depends on the order in which the buttons are released. If the *stop* button is released first, the motor starts. Otherwise the motor does not start. Modify the state diagram that you prepared in Exercise 5.6 to accommodate *start* and *stop* buttons.

5.8 (5) Prepare a state diagram for selecting and dragging objects with the diagram editor described in Exercise 4.2.

A cursor on the diagram tracks a two-button mouse. If the left button is pressed with the cursor on an object (a box or a line), the object is selected, replacing any previously selected object. If the left button is pressed with the cursor not on an object, the selection is set to null. Moving the mouse with the left button held down drags any selected object.

5.9 (6) Extend the diagram editor from Exercise 5.8. If the user left clicks on an object and holds the shift key, the object is added to the selection. Moving the mouse with the left button held down drags any selected objects.

5.10 (5) Figure E5.4 shows a state diagram for a copy machine. Initially the copy machine is off. When power is turned on, the machine reverts to a default state—one copy, automatic contrast and normal size. While the machine is warming, it flashes the ready light. When the machine completes internal testing, the ready light stops flashing and remains on. Then the machine is ready for copying.

The operator may change any of the parameters when the machine is ready. The operator may increment or decrement the number of copies, change the size, toggle between automatic and manual contrast, and change the contrast when auto contrast is disabled. When the parameters are properly set, the operator pushes the start button to begin making copies. Ordinarily copying proceeds until all copies are made. Occasionally the machine may jam or run out

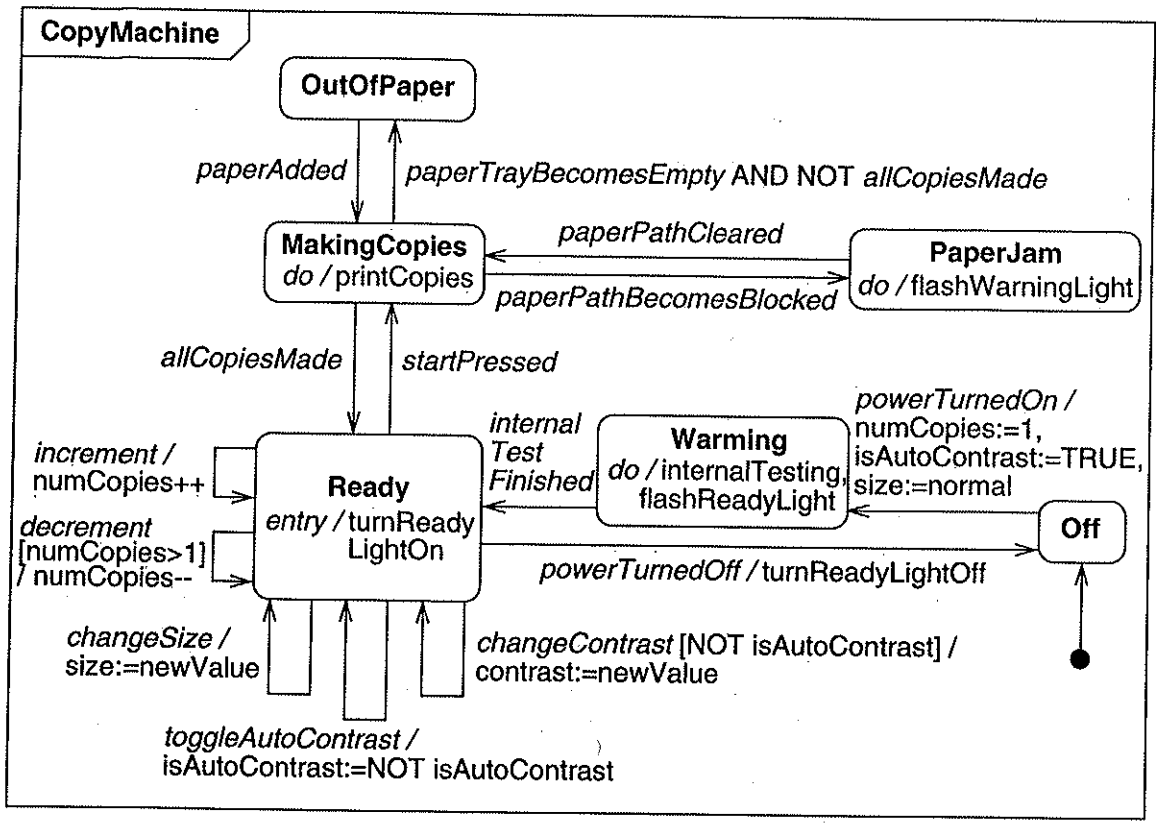


**Figure E5.4 State diagram for a copy machine**

paper. When the machine jams, the operator may clear the blockage and the machine will resume making copies. Adding paper allows the machine to proceed after running out of paper.

Extend the diagram for the following observations. The copy machine does not work quite right. When it jams, the operator must first remove the jammed paper and then turn the machine off and on before it will operate correctly again. If the machine is turned off and on without first removing the offending paper, the machine stays jammed.

5.11 (7) While exploring an old castle, you and a friend discovered a bookcase that you suspected to be the entrance to a secret passageway. While you examined the bookcase, your friend removed a candle from its holder, only to discover that the candle holder was the entrance control. The bookcase rotated a half turn, pushing you along, separating you from your friend. Your friend put the candle back. This time the bookcase rotated a full turn, still leaving you behind it.

Your friend took the candle out. The bookcase started to rotate a full turn again, but this time you stopped it just shy of a full turn by blocking it with your body. Your friend handed you the candle and together you managed to force the bookcase back a half turn, but this left your friend behind it and you in front of it. You put the candle back. As the bookcase began to rotate, you took out the candle, and the bookcase stopped after a quarter turn. You and your friend then entered to explore further.

Prepare a state diagram for the control of the bookcase that is consistent with the previous scenario. What should you have done at first to gain entry with the least fuss?