3.73 Statechart Diagram

3.73.1 Semantics

Statechart diagrams represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances. Typically, it is used for describing the behavior of classes, but statecharts may also describe the behavior of other model entities such as use-cases, actors, subsystems, operations, or methods.

3.73.2 Notation

A statechart diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that inter-connect them. States may also contain subdiagrams by physical containment or tiling. Note that every state machine has a top state, which contains all the other elements of the entire state machine. The graphical rendering of this top state is optional.

The association between a state machine and its context does not have a special notation.

An example statechart diagram for a simple telephone object is depicted in Figure 3-59 on page 3-127.

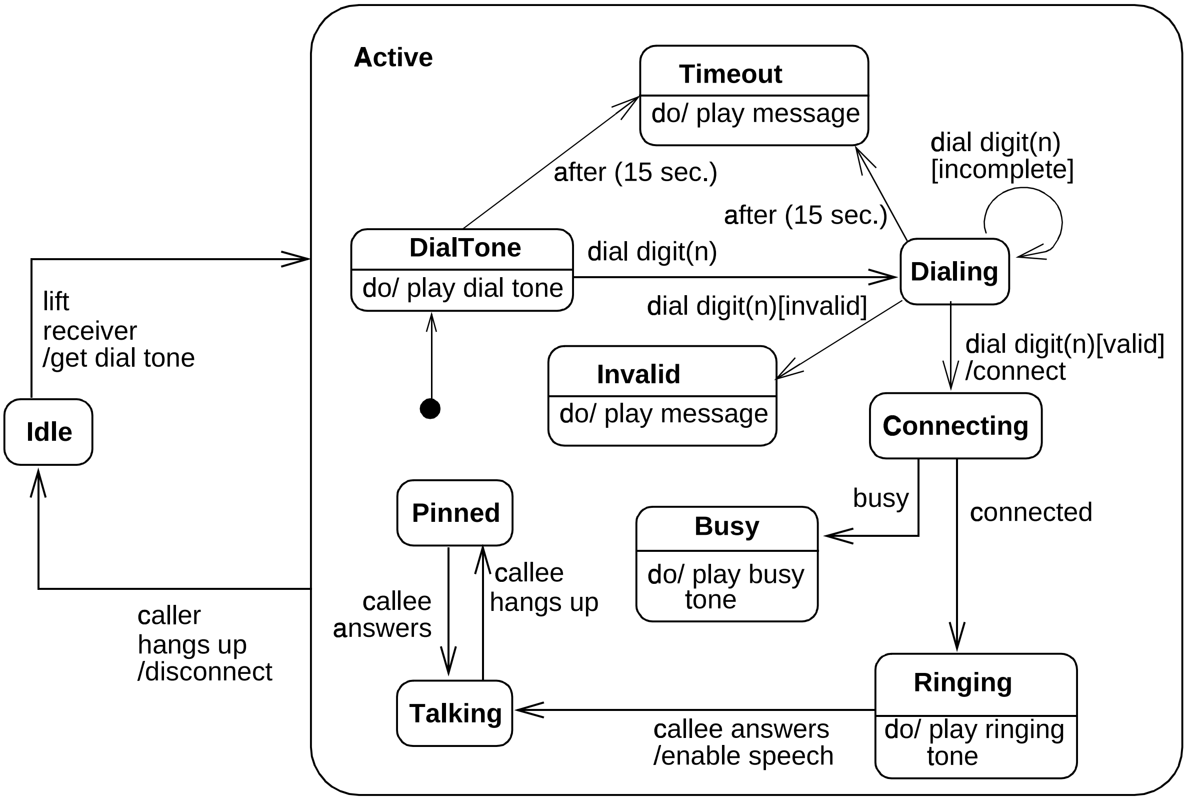

Figure 3-59 State Diagram

3.73.3 Mapping

A statechart diagram maps into a StateMachine. That StateMachine may be owned by a model element capable of dynamic behavior, such as classifier or a behavioral feature, which provides the context for that state machine. Different contexts may apply different semantic constraints on the state machine.

3.74 State

3.74.1 Semantics

A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. A *composite* state is a state that, in contrast to a *simple* state, has a graphical decomposition. (Composite states and their notation are described in more detail in Section 3.75, “Composite States,” on page 3-130.) Conceptually, an object remains in a state for an interval of time. However, the semantics allow for modeling “flow-through” states that are instantaneous, as well as transitions that are not instantaneous.

A state may be used to model an ongoing activity. Such an activity is specified either by a nested state machine or by a computational expression.

3.74.2 Notation

A state is shown as a rectangle with rounded corners (Figure 3-60 on page 3-129). Optionally, it may have an attached name tab. The name tab is a rectangle, usually resting on the outside of the top side of a state and it contains the name of that state. It is normally used to keep the name of a composite state that has concurrent regions, but may be used in other cases as well (the Process state in Figure 3-65 on page 3-137 illustrates the use of the name tab).

A state may be optionally subdivided into multiple compartments separated from each other by a horizontal line. They are as follows:

- Name compartment

  This compartment holds the (optional) name of the state, as a string. States without names are anonymous and are all distinct. It is undesirable to show the same named state twice in the same diagram, as confusion may ensue. Name compartments should not be used if a name tab is used and vice versa.

- Internal transitions compartment

  This compartment holds a list of internal actions or activities that are performed while the element is in the state. The notation for each of these list items has the following general format:

  action-label '/' *action-expression*

The action label identifies the circumstances under which the action specified by the action expression will be invoked. The action expression may use any attributes and links that are in the scope of the owning entity. For list items where the action expression is empty, the backslash separator is optional.

A number of action labels are reserved for various special purposes and cannot be used as event names. The following are the reserved action labels and their meaning:

- entry

  This label identifies an action, specified by the corresponding action expression, which is performed upon entry to the state (entry action).

- exit

  This label identifies an action, specified by the corresponding action expression, that is performed upon exit from the state (exit action).

- do

  This label identifies an ongoing activity ("do activity") that is performed as long as the modeled element is in the state or until the computation specified by the action expression is completed (the latter may result in a completion event being generated).

- include

  This label is used to identify a submachine invocation. The action expression contains the name of the submachine that is to be invoked. Submachine states and the corresponding notation are described in Section 3.81, “Submachine States,” on page 3-142.

In all other cases, the action label identifies the event that triggers the corresponding action expression. These events are called internal transitions and are semantically equivalent to self transitions *except that the state is not exited or re-entered*. This means that the corresponding exit and entry actions are not performed. The general format for the list item of an internal transition is:

> *event-name* ‘(’ *comma-separated-parameter-list* ‘)’ ‘[’ *guard-condition*‘]’ ‘/’ *action-expression*

Each event name may appear more than once per state if the guard conditions are different. The event parameters and the guard conditions are optional. If the event has parameters, they can be used in the action expression through the current event variable.

3.74.3 Example

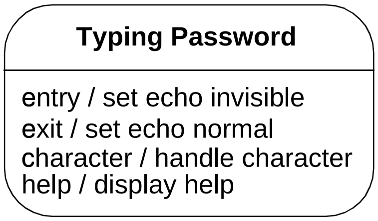

Figure 3-60 State

3.74.4 Mapping

A state symbol maps into a State. See Section 3.75, “Composite States,” on page 3-130 for further details on which kind of state.

The name string in the symbol maps to the name of the state. Two symbols with the same name map into the same state . However, each state symbol with no name (or an empty name string) maps into a distinct anonymous State.

A list item in the internal transition compartment maps into a corresponding Action associated with a state. An “entry” list item (i.e., an item with the “entry” label) maps to the “entry” role, an “exit” list item maps to the “exit” role, and a “do” item maps to the “doActivity” role. (The mapping of “include” items is discussed in Section 3.81, “Submachine States,” on page 3-142.)

A list item with an event name maps to a Transition associated with the “internal” role relative to the state. The action expression maps into the ActionSequence and Guard for the Transition. The event name and arguments map into an Event corresponding to the event name and arguments. The Transition has a *trigger* Association to the Event.

3.75 Composite States

3.75.1 Semantics

A composite state is decomposed into two or more concurrent substates (called *regions*) or into mutually exclusive disjoint substates. A given state may only be refined in one of these two ways. Naturally, any substate of a composite state can also be a composite state of either type.

A newly-created object takes its topmost default transition, originating from the topmost initial pseudostate. An object that transitions to its outermost final state is terminated.

Each region of a state may have initial pseudostates and final states. A transition to the enclosing state represents a transition to the initial pseudostate. A transition to a final state represents the completion of activity in the enclosing region. Completion of activity in all concurrent regions represents completion of activity by the enclosing state and triggers a completion event on the enclosing state. Completion of the top state of an object corresponds to its termination.

3.75.2 Notation

An expansion of a state shows its internal state machine structure. In addition to the (optional) name and internal transition compartments, the state may have an additional compartment that contains a region holding a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic region.

An expansion of a state into concurrent substates is shown by tiling the graphic region of the state using dashed lines to divide it into regions. Each region is a concurrent substate. Each region may have an optional name and must contain a nested state diagram with disjoint states. The text compartments of the entire state are separated from the concurrent substates by a solid line. It is also possible to use a tab notation to place the name of a concurrent state. The tab notation is more space efficient.

An expansion of a state into disjoint substates is shown by showing a nested state diagram within the graphic region.

An initial pseudostate is shown as a small solid filled circle. In a top-level state machine, the transition from an initial pseudostate may be labeled with the event that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any transition to the enclosing state. The initial transition may have an action.

A final state is shown as a circle surrounding a small solid filled circle (a bull's eye). It represents the completion of activity in the enclosing state and it triggers a transition on the enclosing state labeled by the implicit activity completion event (usually displayed as an unlabeled transition), if such a transition is defined.

In some cases, it is convenient to hide the decomposition of a composite state. For example, the state machine inside a composite state may be very large and may simply not fit in the graphical space available for the diagram. In that case, the composite state may be represented by a simple state graphic with a special "composite" icon, usually in the lower right-hand corner. This icon, consisting of two horizontally placed and connected states, is an *optional* visual cue that the state has a decomposition that is not shown in this particular statechart diagram (Figure 3-62 on page 3-131). Instead, the contents of the composite state are shown in a separate diagram. Note that the "hiding" here is purely a matter of graphical convenience and has no semantic significance in terms of access restrictions.

3.75.3 Examples

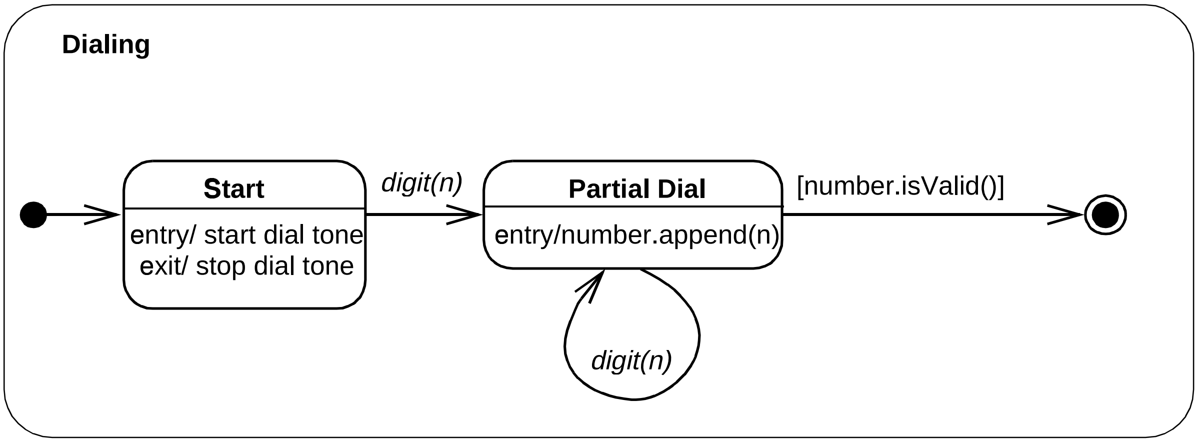

Figure 3-61 Sequential Substates

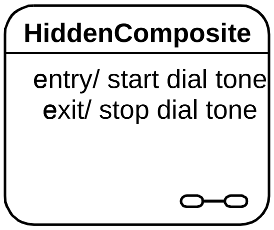

Figure 3-62 Composite State with hidden decomposition indicator icon

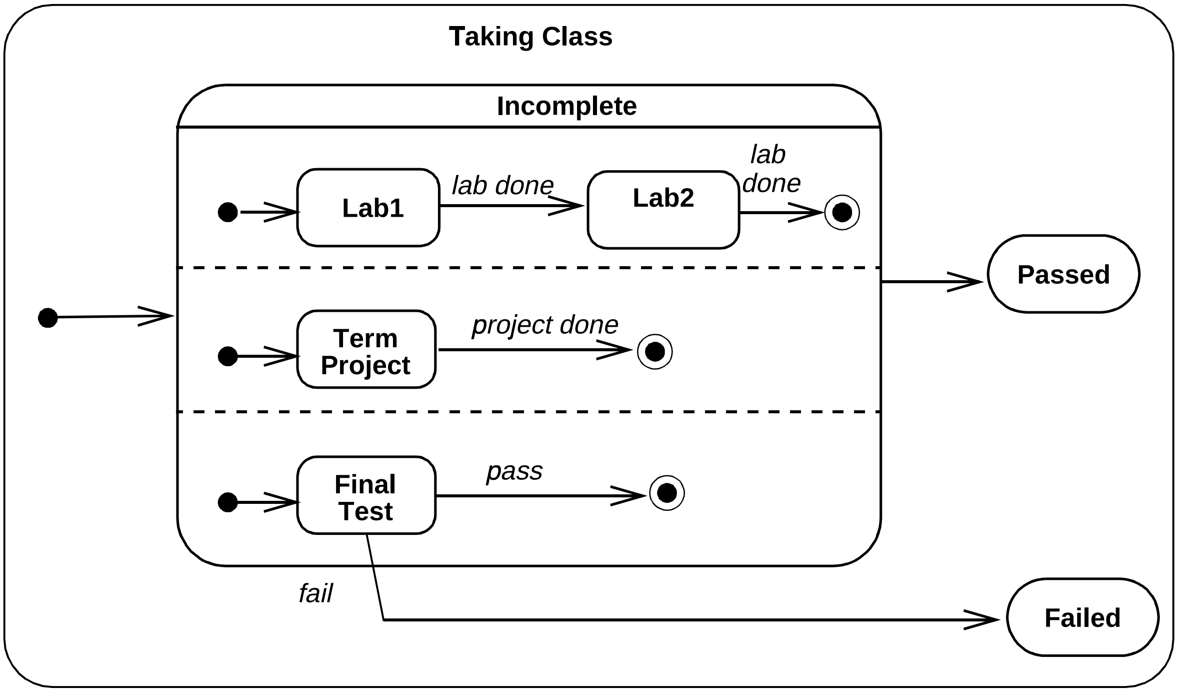

Figure 3-63 Concurrent Substates

3.75.4 Mapping

A state symbol maps into a State. If the symbol has no subdiagrams in it, it maps into a SimpleState. If it is tiled by dashed lines into regions, then it maps into a CompositeState with the *isConcurrent* value true; otherwise, it maps into a CompositeState with the *isConcurrent* value false. A region maps into a CompositeState with the *isRegion* value true and the *isConcurrent* value false.

An initial pseudostate symbol map into a Pseudostate of kind *initial.* A final state symbol maps to a *final* state.

3.76 Events

3.76.1 Semantics

An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. Events may be of several kinds (not necessarily mutually exclusive).

- A designated condition becoming true (described by a Boolean expression) results in a change event instance. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition. A guard condition is evaluated *once* whenever its event fires. If it is false, then the transition does not occur and the event is lost.
- The receipt of an explicit signal from one object to another results in a signal event instance. It is denoted by the signature of the event as a trigger on a transition.
- The receipt of a call for an operation implemented as a transition by an object represents a call event instance.
- The passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time is a TimeEvent.

The event declaration has scope within the package it appears in and may be used in state diagrams for classes that have visibility inside the package. An event is *not* local to a single class.

3.76.2 Notation

A signal or call event can be defined using the following format:

event-name ‘(‘ *comma-separated-parameter-list* ‘)

A parameter has the format:

parameter-name ‘:’ *type-expression*

A signal can be declared using the «signal» keyword on a class symbol in a class diagram. The parameters are specified as attributes. A signal can be specified as a subclass of another signal. This indicates that an occurrence of the subevent triggers any transition that depends on the event or any of its ancestors.

An elapsed-time event can be specified with the keyword **after** followed by an expression that evaluates (at modeling time) to an amount of time, such as “**after** (5 seconds)” or **after** (10 seconds since exit from state A).” If no starting point is indicated, then it is the time since the entry to the current state. Other time events can be specified as conditions, such as **when** (date = Jan. 1, 2000).

A condition becoming true is shown with the keyword **when** followed by a Boolean expression. This may be regarded as a continuous test for the condition until it is true, although in practice it would only be checked on a change of values.

Signals can be declared on a class diagram with the keyword «signal» on a rectangle symbol. These define signal names that may be used to trigger transitions. Their parameters are shown in the attribute compartment. They have no operations. They may appear in a generalization hierarchy.

3.76.3 Example

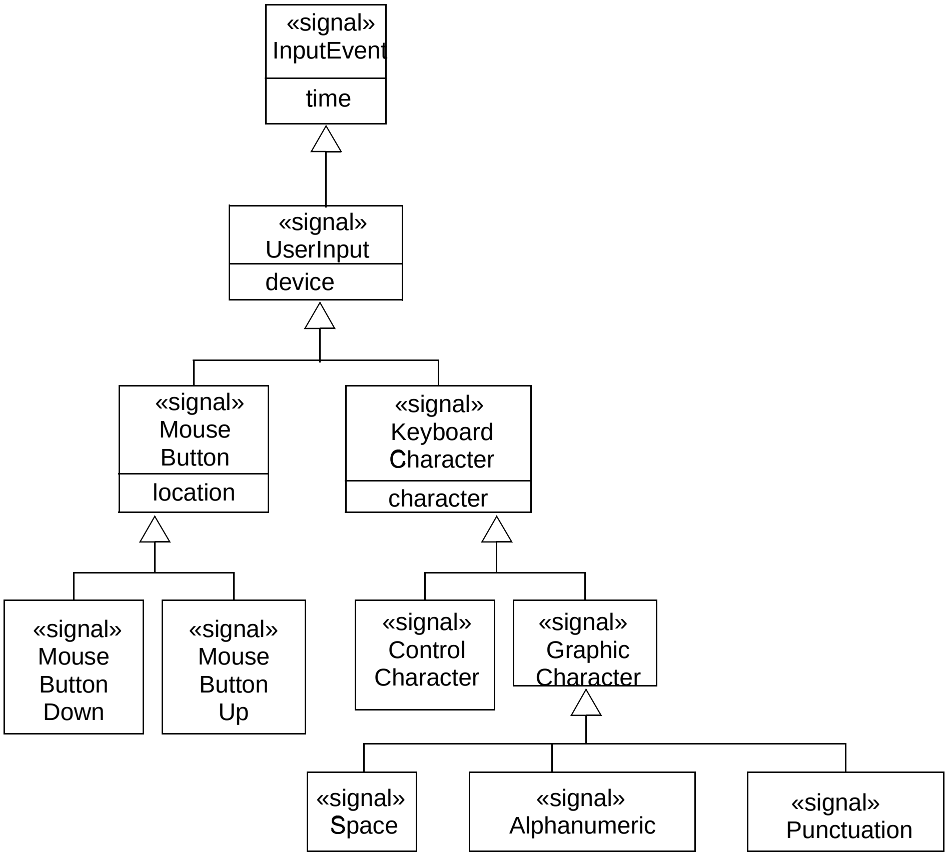

Figure 3-64 Signal Declaration

3.76.4 Mapping

A class box with stereotype «signal» maps into a Signal. The name and parameters are given by the name string and the attribute list of the box. Generalization arrows between signal class boxes map into Generalization relationships between the Signal.

The usage of an event string expression in a context requiring an event maps into an implicit reference of the Event with the given name. It is an error if various uses of the same name (including any explicit declarations) do not match.

3.77 Simple Transitions

3.77.1 Semantics

A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain specified conditions are satisfied. On such a change of state, the transition is said to "fire." The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. Events are processed one at a time. If an event does not trigger any transition, it is discarded. If it can trigger more than one transition within the same sequential region (i.e., not in different concurrent regions), only one will fire. If these conflicting transitions are of the same priority, an arbitrary one is selected and triggered.

3.77.2 Notation

A transition is shown as a solid line originating from the *source* state and terminated by an arrow on the *target* state. It may be labeled by a *transition string* that has the following general format:

event-signature '[' guard-condition ']' '/' *action-expression*

The *event-signature* describes an event with its arguments:

event-name '(' *comma-separated-parameter-list* ')'

The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. The guard condition may also involve tests of concurrent states of the current machine, or explicitly designated states of some reachable object (for example, "**in** State1" or "**not in** State2"). State names may be fully qualified by the nested states that contain them, yielding pathnames of the form "State1::State2::State3." This may be used in case same state name occurs in different composite state regions of the overall machine.

The *action-expression* is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the owning object and the parameters of the triggering event, or any other features visible in its scope. The corresponding action must be executed entirely before any other actions are considered. This model of execution is referred to as *run-to-completion* semantics. The action expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending signals or invoking operations. The details of this expression are dependent on the action language chosen for the model.

3.77.2.1 Transition times

Names may be placed on transitions to designate the times at which they fire. See Section 3.62, "Transition Times," on page 3-104.

3.77.3 Example

right-mouse-down (location) [location in window] / object := pick-object (location); object.highlight ()

The event may be any of the standard event types. Selecting the type depends on the syntax of the name (for time events, for example); however, SignalEvents and CallEvents are not distinguishable by syntax and must be discriminated by their declaration elsewhere.

3.77.4 Mapping

A transition string and the transition arrow that it labels together map into a Transition and its attachments. The arrow connects two state symbols. The Transition has the corresponding States as its source (the state at the tail) and destination (the state at the head) States in association to the Transition.

The event name and parameters map into an Event element, which may be a SignalEvent, a CallEvent, a TimeExpression (if it has the proper syntax), or a ChangeEvent (if it is expressed as a Boolean expression). The event is attached as a "trigger" role in the association to the transition.

The guard condition maps into a Guard element attached to the Transition. Note that a guard condition is distinguished graphically from a change event specification by being enclosed in brackets.

An action expression maps into an Action attached as an "effect" role relative to the Transition.

3.78 Transitions to and from Concurrent States

A concurrent transition may have multiple source states and target states. It represents a synchronization and/or a splitting of control into concurrent threads without concurrent substates.

3.78.1 Semantics

A concurrent transition is enabled when all the source states are occupied. After a compound transition fires, all its destination states are occupied.

3.78.2 Notation

A concurrent transition includes a short heavy bar (a *synchronization* bar, which can represent synchronization, forking, or both). The bar may have one or more arrows from states to the bar (these are the *source states*). The bar may have one or more arrows from the bar to states (these are the *destination states*). A transition string may be shown near the bar. Individual arrows do not have their own transition strings.

3.78.3 Example

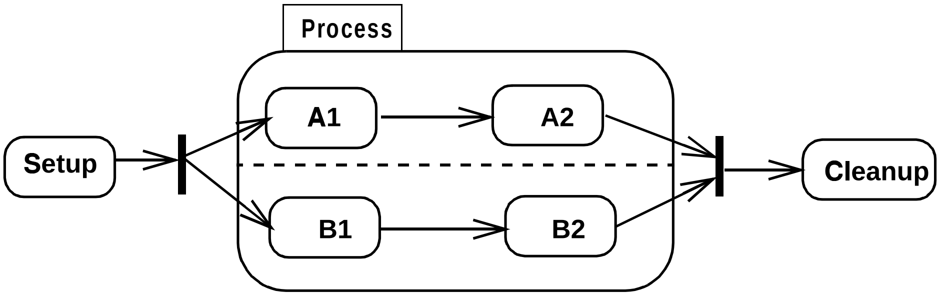

Figure 3-65 Concurrent Transitions

3.78.4 Mapping

A bar with multiple transition arrows leaving it maps into a fork pseudostate. A bar with multiple transition arrows entering it maps into a join pseudostate. The transitions corresponding to the incoming and outgoing arrows attach to the pseudostate as if it were a regular state. If a bar has multiple incoming and multiple outgoing arrows, then it maps into a join connected to a fork pseudostate by a single transition with no attachments.

3.79 Transitions to and from Composite States

3.79.1 Semantics

A transition drawn to the boundary of a composite state is equivalent to a transition to its initial point (or to a complex transition to the initial point of each of its concurrent regions, if it is concurrent). The entry action is always performed when a state is entered from outside.

A transition from a composite state indicates a transition that applies to each of the states within the state region (at any depth). It is “inherited” by the nested states. Inherited transitions can be masked by the presence of nested transitions with the same trigger.

3.79.2 Notation

A transition drawn to a composite state boundary indicates a transition to the composite state. This is equivalent to a transition to the initial pseudostate within the composite state region. The initial pseudostate must be present. If the state is a concurrent composite state, then the transition indicates a transition to the initial pseudostate of each of its concurrent substates.

Transitions may be drawn directly to states within a composite state region at any nesting depth. All entry actions are performed for any states that are entered on any transition. On a transition within a concurrent composite state, transition arrows from the synchronization bar may be drawn to one or more concurrent states. Any other concurrent regions start with their default initial pseudostate.

A transition drawn from a composite state boundary indicates a transition of the composite state. If such a transition fires, any nested states are forcibly terminated and perform their exit actions, then the transition actions occur and the new state is established.

Transitions may be drawn directly from states within a composite state region at any nesting depth to outside states. All exit actions are performed for any states that are exited on any transition. On a transition from within a concurrent composite state, transition arrows may be specified from one or more concurrent states to a synchronization bar; therefore, specific states in the other regions are irrelevant to triggering the transition.

A state region may contain a *history state indicator* shown as a small circle containing an ‘H.’ The history indicator applies to the state region that directly contains it. A history indicator may have any number of incoming transitions from outside states. It may have at most one outgoing unlabeled transition. This identifies the default “previous state” if the region has never been entered. If a transition to the history indicator fires, it indicates that the object resumes the state it last had within the composite region. Any necessary entry actions are performed. The history indicator may also be ‘H*’ for *deep history.* This indicates that the object resumes the state it last had at any depth within the composite region, rather than being restricted to the state at the same level as the history indicator. A region may have both shallow and deep history indicators.

3.79.3 Presentation Options

3.79.3.1 Stubbed transitions

Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state. Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial pseudostate may (but need not) be shown as coming from or going to *stubs*. A *stub* is shown as a small vertical line (bar) drawn inside the boundary of the enclosing state. It indicates a transition connected to a suppressed internal state. Stubs are not used for transitions to initial or from final states.

Note that events should be shown on transitions leading into a state, either to the state boundary or to an internal substate, including a transition to a stubbed state. Normally events should not be shown on transitions leading from a stubbed state to an external state. Think of a transition as belonging to its source state. If the source state is suppressed, then so are the details of the transition. Note also that a transition from a final state is summarized by an unlabeled transition from the composite state contour (denoting the implicit event “action complete” for the corresponding state).

3.79.4 Example

See Figure 3-64 on page 3-134 and Figure 3-65 on page 3-137 for examples of composite transitions. The following are examples of stubbed transitions and the history indicator.

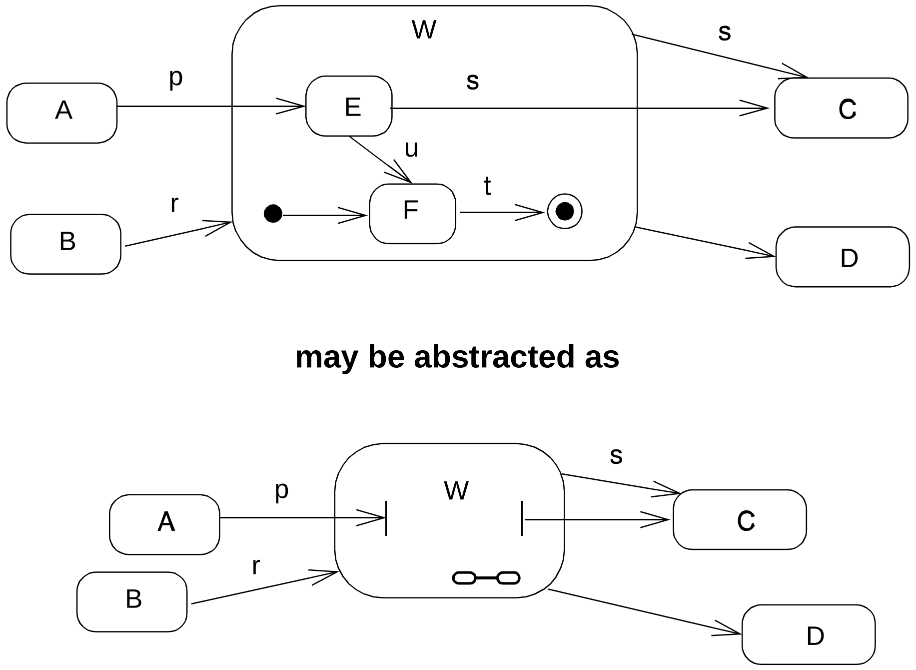

Figure 3-66 Stubbed Transitions

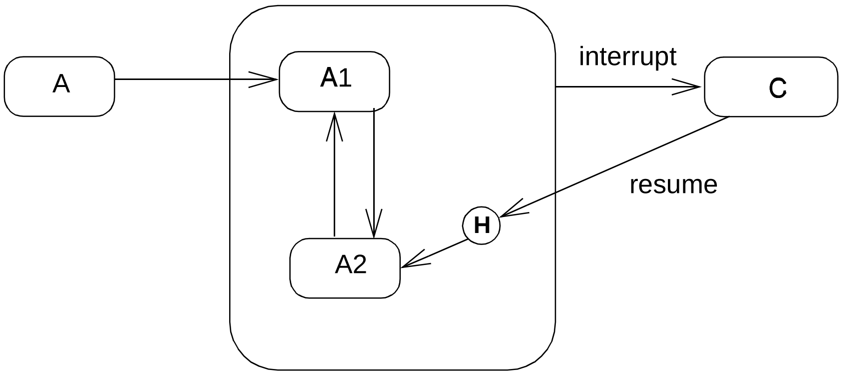

Figure 3-67 History Indicator

3.79.5 Mapping

An arrow to any state boundary, nested or not, maps into a Transition between the corresponding States and similarly for transitions directly to history states.

A history indicator maps into a Pseudostate of kind *shallowHistory* or *deepHistory*.

A stubbed transition does not map into anything in the model. It is a notational elision that indicates the presence of transitions to additional states in the model that are not visible in the diagram.

3.80 Factored Transition Paths

3.80.1 Semantics

By definition, a transition connects exactly two vertices in the state machine graph. However, since some of these vertices may be pseudostates (which are transient in nature) there is a need for describing chains of transitions that may be executed in the context of a single run-to-completion step. Such a transition is known as a *compound transition*.

As a practical measure, it is often useful to share segments of a compound transition. For example, two or more distinct compound transitions may come together and continue via a common path, sharing its action, and possibly terminating on the same target state. In other cases, it may be useful to split a transition into separate mutually exclusive (i.e., non-concurrent) paths.

Both of these examples of graphical factoring in which some transitions are shared result in simplified diagrams. However, factoring is also useful for modeling dynamically adaptive behavior. An example of this occurs when a single event may lead to any of a set of possible target states, but where the final target state is only determined as the result of an action (calculation) performed after the triggering of the compound transition.

Note that the splitting and joining of paths due to factoring is different from the splitting and joining of concurrent transitions described in Section 3.78, “Transitions to and from Concurrent States,” on page 3-136. The sources and targets of these factored transitions are not concurrent.

3.80.2 Notation

Two or more transitions emanating from different non-concurrent states or pseudostates can terminate on a common junction point. This allows their respective compound transitions to share the path that emanates from that junction point. A junction point is represented by a small black circle. Alternatively, it may be represented by a diamond shape (see Section 3.86, “Decisions,” on page 3-150).

Two or more guarded transitions emanating from the same junction point represent a *static branch point*. Normally, the guards are mutually exclusive. This is equivalent to a set of individual transitions, one for each path through the tree, whose guard condition is the "and" of all of the conditions along the path. Note that the semantics of static branches is that all the outgoing guards are evaluated *before* any transition is taken.

Two or more guarded transitions emanating from a common *dynamic choice point* are used to model dynamic choices. In this case, the guards of the outgoing transitions are evaluated at the time the choice point has been reached. The value of these guards may be a function of some calculations performed in the actions of the incoming transition(s). A dynamic choice point is represented by a small white circle (reminiscent of a small state icon).

3.80.3 Examples

In Figure 3-68 a single junction point is used to merge and split transitions. Regardless of whether the junction point was reached from state State0 or from state State1, the outgoing paths are the same for both cases.

If the state machine in this example is in state State1 and b is less than 0 when event e1 occurs, the outgoing transition will be taken only if one of the three downstream guards is true. Thus, if a is equal to 6 at that point, no transition will be triggered.

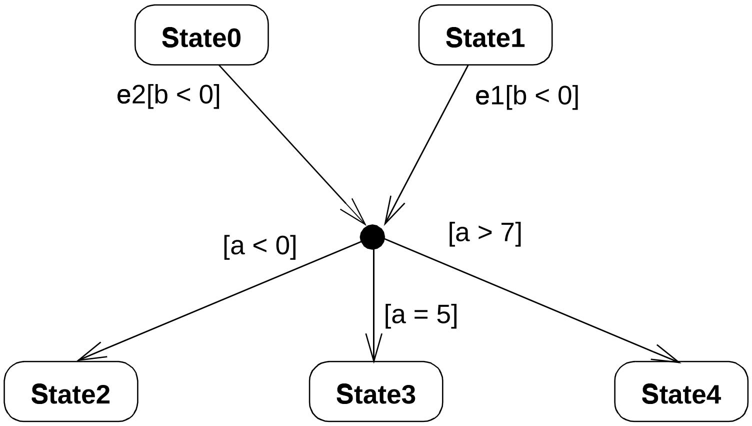

Figure 3-68 Junction points

In the dynamic choice point example in Figure 3-69 on page 3-142, the decision on which branch to take is only made after the transition from State1 is taken and the choice point is reached. Note that the action associated with that incoming transition computes a new value for a. This new value can then be used to determine the outgoing transition to be taken. The use of the predefined condition[else] is recommended to avoid run-time errors.

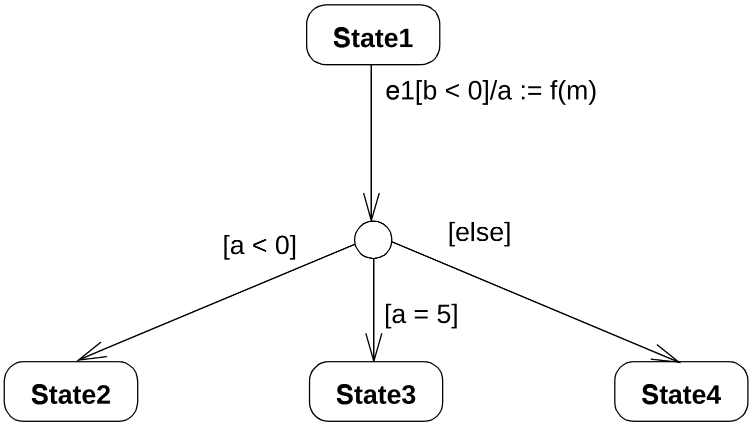

Figure 3-69 Dynamic choice points

3.81 Submachine States

3.81.1 Semantics

A submachine state represents the *invocation* of a state machine defined elsewhere. It is similar to a macro call in the sense that it represents a (graphical) shorthand that implies embedding of a complex specification within another specification. The submachine must be contained in the same context as the invoking state machine.

In the general case, an invoked state machine can be entered at any of its substates or through its default (initial) pseudostate. Similarly, it can be exited from any substate or as a result of the invoked state machine reaching its final state or by an "inherited" or "group" transition that applies to all substates in the submachine.

The non-default entry and exits are specified through special stub states.

3.81.2 Notation

The submachine state is depicted as a normal state with the appropriate "include" declaration within its internal transitions compartment (see Section 3.74, "State," on page 3-127). The expression following the include reserved word is the name of the invoked submachine.

Optionally, the submachine state may contain one or more entry stub states and one or more exit stub states. The notation for these is similar to that used for stub ends of stubbed transitions, except that the ends are labeled. The labels represent the names of the corresponding substates within the invoked submachine. A pathname may be used if the substate is not defined at the top level of the invoked submachine. Naturally, this name must be a valid name of a state in the invoked state machine.

If the submachine is entered through its default pseudostate or if it is exited as a result of the completion of the submachine, it is not necessary to use the stub state notation for these cases. Similarly, a stub state is not required if the exit occurs through an explicit "group" transition that emanates from the boundary of the submachine state (implying that it applies to all the substates of the submachine).

Submachine states invoking the same submachine may occur multiple times in the same state diagram with different entry and exit configurations and with different internal transitions and exit and entry action specifications in each case.

3.81.3 Example

The following diagram shows a fragment from a statechart diagram in which a submachine (the FailureSubmachine) is invoked in a particular way. The actual submachine is presumably defined elsewhere and is not shown in this diagram. Note that the same submachine could be invoked elsewhere in the same state chart diagram with different entry and exit configurations.

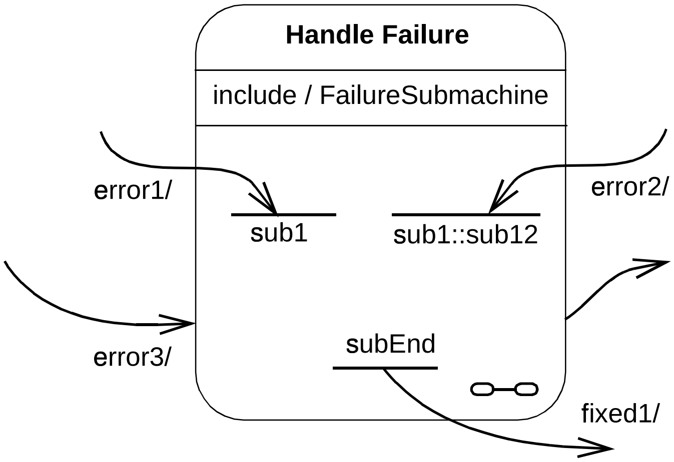

Figure 3-70 Submachine State

In the above example, the transition triggered by event "error1" will terminate on state "sub1" of the FailureSubmachine state machine. Since the entry point does not contain a path name, this means that "sub1" is defined at the top level of that submachine. In contrast, the transition triggered by "error2" will terminate on the "sub12" substate of the "sub1"substate (as indicated by the path name), while the "error3" transition implies taking of the default transition of the FailureSubmachine.

The transition triggered by the event "fixed1" emanates from the "subEnd" substate of the submachine. Finally, the transition emanating from the edge of the submachine state is taken as a result of the completion event generated when the FailureSubmachine reaches its final state.

3.81.4 Mapping

A submachine state in a statechart diagram maps directly to a SubmachineState in the metamodel. The name following the “include” reserved action label represents the state machine indicated by the “submachine” attribute. Stub states map to the Stub State concept in the metamodel. The label on the diagram corresponds to the pathname represented by the “referenceState” attribute of the stub state.

3.82 Synch States

3.82.1 Semantics

A synch state is for synchronizing concurrent regions of a state machine. It is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states. The firing of outgoing transitions from a synch state can be limited by specifying a bound on the difference between the number of times outgoing and incoming transitions have fired.

3.82.2 Notation

A synch state is shown as a small circle with the upper bound inside it. The bound is either a positive integer or a star ('*') for unlimited. Synch states are drawn on the boundary between two regions when possible.

3.82.3 Example

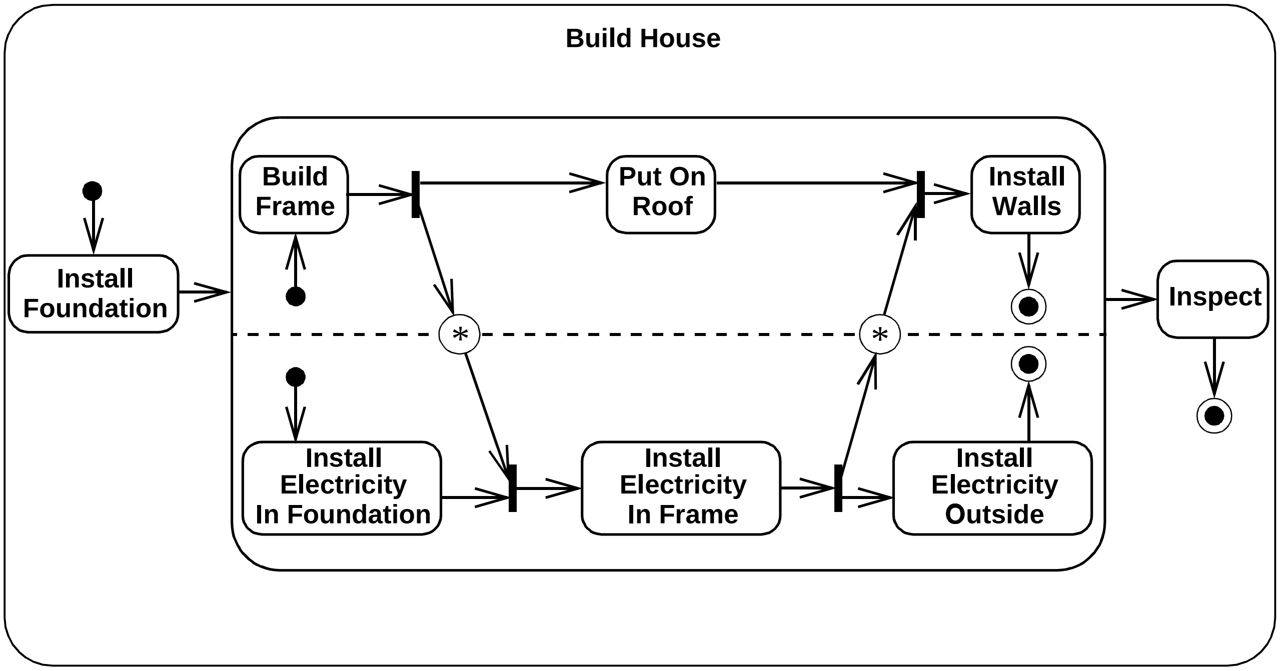

Figure 3-71 Synch states

3.82.4 Mapping

A synch state circle maps into a SynchState, contained by the least common containing state of the regions it is synchronizing. The number inside it maps onto the bound attribute of the synch state. A star ('*') inside the synch state circle maps to a value of Unlimited for the bound attribute.

Part 10 - Activity Diagrams

3.83 Activity Diagram

3.83.1 Semantics

An activity graph is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. It represents a state machine of a procedure itself.

3.83.2 Notation

An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states. The entire activity diagram is attached (through the model) to a class, such as a use case, or to a package, or to the implementation of an operation. The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions (that is, procedural flow of control). Use ordinary state diagrams in situations where asynchronous events occur.

3.83.3 Example

Person::Prepare Beverage

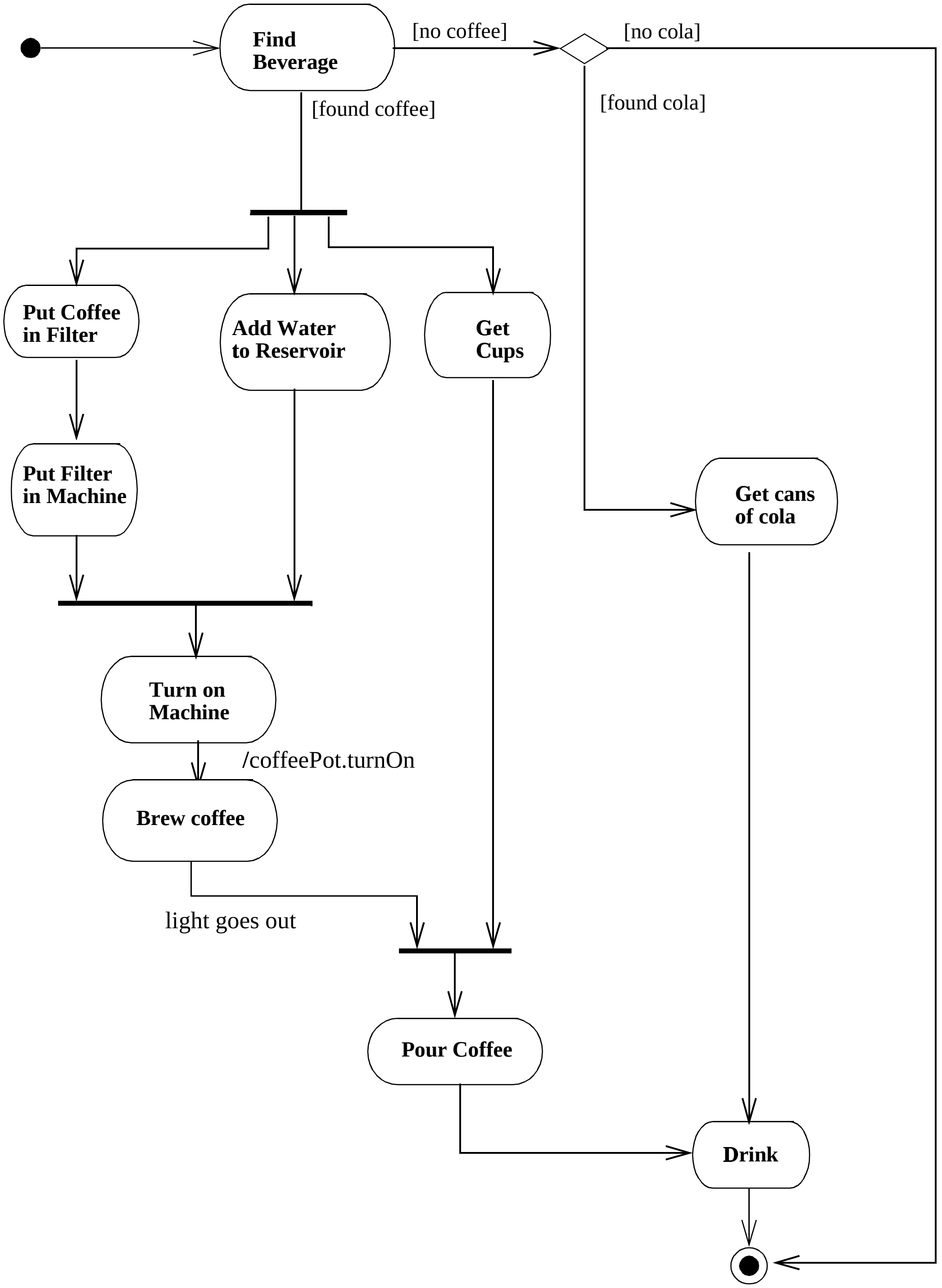

Figure 3-72 Activity Diagram

3.83.4 Mapping

An activity diagram maps into an ActivityGraph.

3.84 Action State

3.84.1 Semantics

An *action state* is shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action (there may be several such transitions if they have guard conditions). Action states should not have internal transitions or outgoing transitions based on explicit events, use normal states for this situation. The normal use of an action state is to model a step in the execution of an algorithm (a procedure) or a workflow process.

3.84.2 Notation

An action state is shown as a shape with straight top and bottom and with convex arcs on the two sides. The *action-expression* is placed in the symbol. The action expression need not be unique within the diagram.

Transitions leaving an action state should not include an event signature. Such transitions are implicitly triggered by the completion of the action in the state. The transitions may include guard conditions and actions.

3.84.3 Presentation options

The action may be described by natural language, pseudocode, or programming language code. It may use only attributes and links of the owning object.

Note that action state notation may be used within ordinary state diagrams; however, they are more commonly used with activity diagrams, which are special cases of state diagrams.

3.84.4 Example

Figure 3-73 Action States

3.84.5 Mapping

An action state symbol maps into an ActionState with the action-expression mapped to the entry action of the State. There is no *exit* nor any internal transitions. The State is normally anonymous.

3.85 Subactivity State

3.85.1 Semantics

A *subactivity state* invokes an activity graph. When a subactivity state is entered, the activity graph “nested” in it is executed as any activity graph would be. The subactivity state is not exited until the final state of the nested graph is reached, or when trigger events occur on transitions coming out of the subactivity state. Since states in activity graphs do not normally have trigger events, subactivity states are normally exited when their nested graph is finished. A single activity graph may be invoked by many subactivity states.

3.85.2 Notation

A subactivity state is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol. The subactivity need not be unique within the diagram.

This notation is applicable to any UML construct that supports “nested” structure. The icon must suggest the type of nested structure.

3.85.3 Example

Figure 3-74 Subactivity States

3.85.4 Mapping

A subactivity state symbol maps into a SubactivityState. The name of the subactivity maps to a submachine link between the SubactivityState and a StateMachine of that name. The SubactivityState is normally anonymous.

3.86 Decisions

3.86.1 Semantics

A state diagram (and by derivation an activity diagram) expresses a decision when guard conditions are used to indicate different possible transitions that depend on Boolean conditions of the owning object. UML provides a shorthand for showing decisions and merging their separate paths back together.

3.86.2 Notation

A decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger. All possible outcomes should appear on one of the outgoing transitions. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.

The same icon can be used to merge decision branches back together, in which case it is called a merge. A merge has two or more incoming arrows and one outgoing arrow.

Note that a chain of decisions may be part of a complex transition, but only the first segment in such a chain may contain an event trigger label. All segments may have guard expressions. The transition coming from a merge may not have a trigger label or guard expressions.

3.86.3 Example

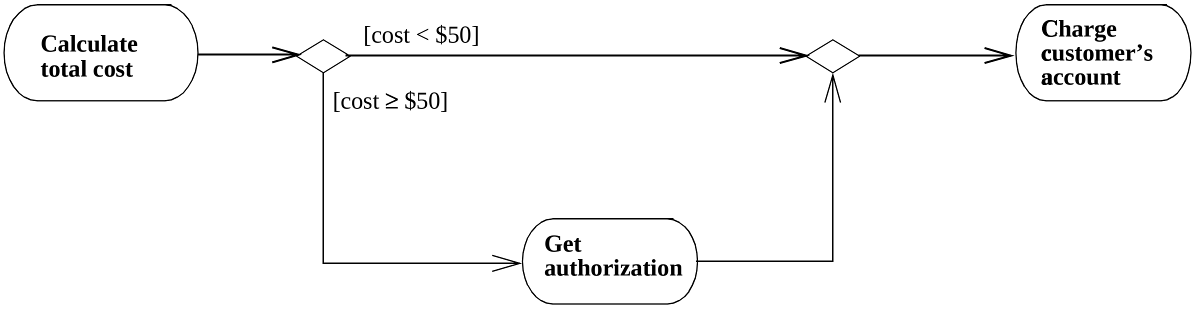

Figure 3-75 Decision and merge

3.86.4 Mapping

A decision symbol maps into a Pseudostate of kind *junction*. Each label on an outgoing arrow maps into a Guard on the corresponding Transition leaving the Pseudostate. A merge symbol also maps into a Pseudostate of kind *junction*.

3.87 Swimlanes

3.87.1 Semantics

Actions and subactivities may be organized into *swimlanes.* Swimlanes are used to organize responsibility for actions and subactivities according to class. They often correspond to organizational units in a business model.

3.87.2 Notation

An activity diagram may be divided visually into “swimlanes,” each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity, and may eventually be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance, but might indicate some affinity. Each action is assigned to one swimlane. Transitions may cross lanes. There is no significance to the routing of a transition path.

3.87.3 Example

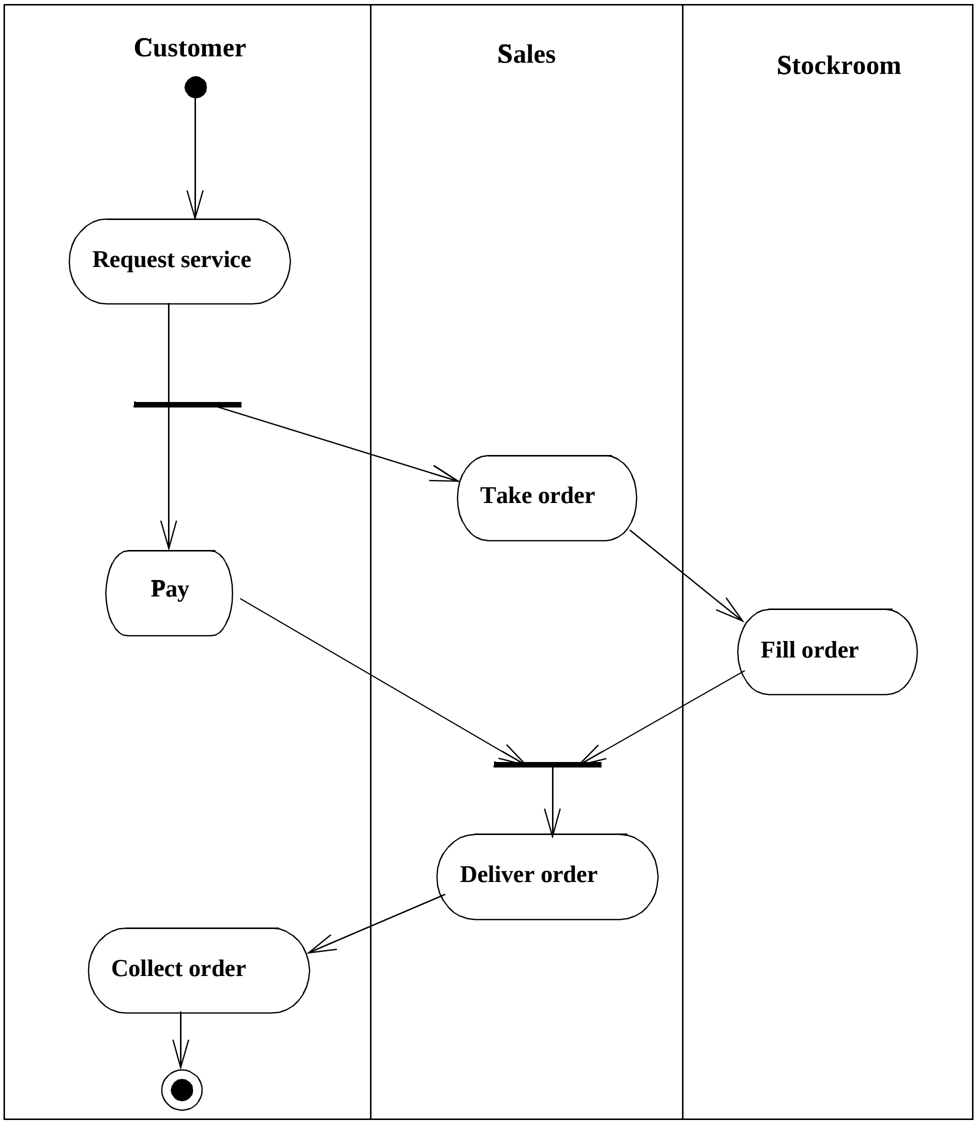

Figure 3-76 Swimlanes in Activity Diagram

3.87.4 Mapping

A swimlane maps into a Partition of the States in the ActivityGraph. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

3.88 Action-Object Flow Relationships

3.88.1 Semantics

Actions operate by and on objects. These objects either have primary responsibility for initiating an action, or are used or determined by the action. Actions usually specify calls sent between the object owning the activity graph, which initiates actions, and the objects that are the targets of the actions.

3.88.2 Notation

3.88.2.1 Object responsible for an action

In sequence diagrams, the object responsible for performing an action is shown by drawing a lifeline and placing actions on lifelines. See Section 3.58, "Sequence Diagram," on page 3-94. Activity diagrams do not show the lifeline, but each action specifies which object performs its operation. These objects may also be related to the swimlane in some way. The actions within a swimlane can all be handled by the same object or by multiple objects.

3.88.2.2 Object flow

Objects that are input to or output from an action may be shown as object symbols. A dashed arrow is drawn from an action state to an output object, and a dashed arrow is drawn from an input object to an action state. The same object may be (and usually is) the output of one action and the input of one or more subsequent actions.

The control flow (solid) arrows must be omitted when the object flow (dashed) arrows supply a redundant constraint. In other words, when a state produces an output that is input to a subsequent state, that object flow relationship implies a control constraint.

3.88.2.3 Object in state

Frequently the same object is manipulated by a number of successive actions or subactivities. It is possible to show one object with arrows to and from all of the relevant actions and subactivities, but for greater clarity, the object may be displayed multiple times on a diagram. Each appearance denotes a different point during the object's life. To distinguish the various appearances of the same object, the state of the object at each point may be placed in brackets and appended to the name of the object (for example, PurchaseOrder[approved]). This notation may also be used in collaboration and sequence diagrams.

3.88.3 Example

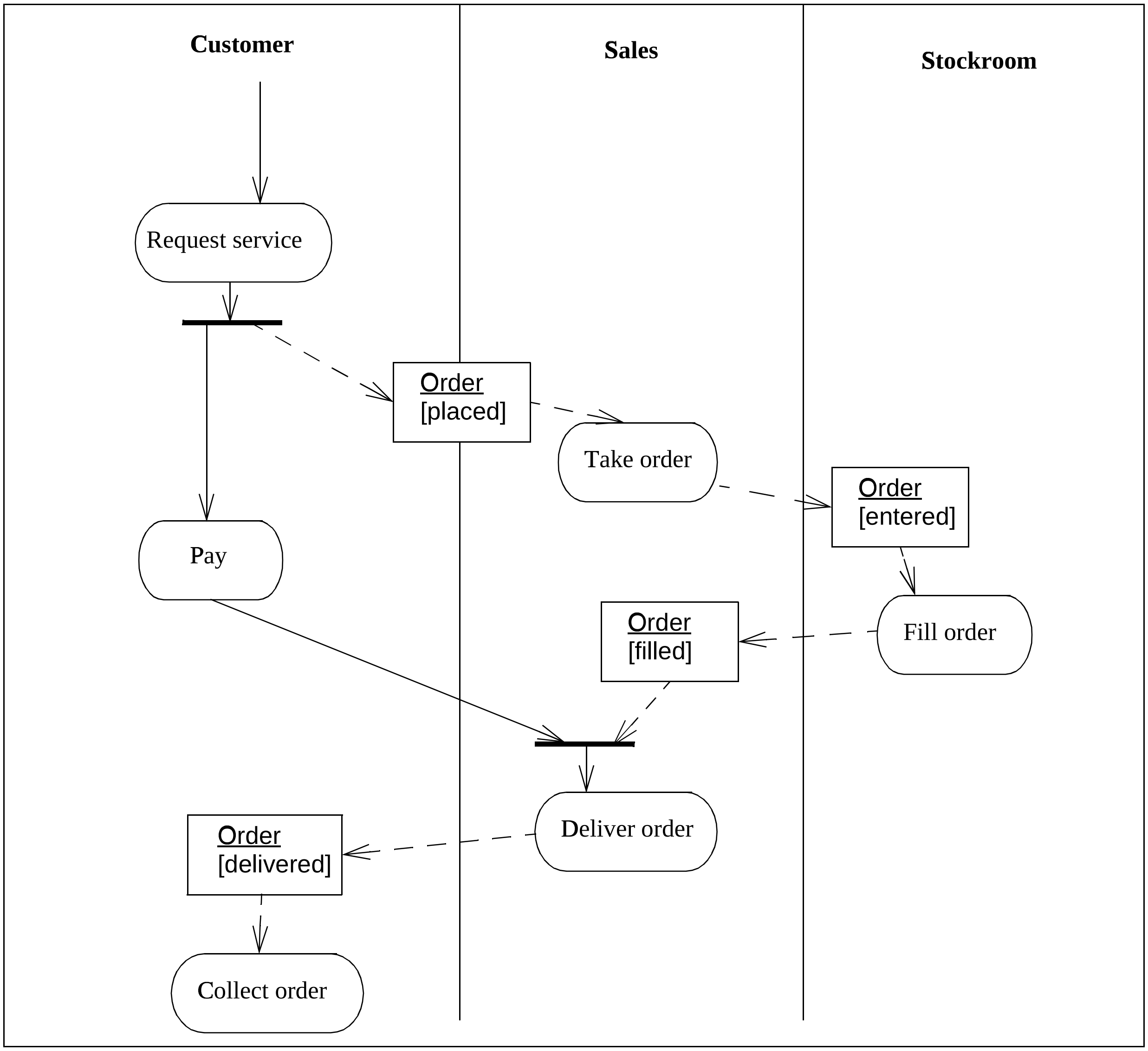

Figure 3-77 Actions and Object Flow

3.88.4 Mapping

An object flow symbol maps into an ObjectFlowState whose incoming and outgoing Transitions correspond to the incoming and outgoing arrows. The Transitions have no attachments. The class name and (optional) state name of the object flow symbol map into a Class or a ClassifierInState corresponding to the name (s). Solid and dashed arrows both map to transitions.

3.89 Control Icons

The following icons provide explicit symbols for certain kinds of information that can be specified on transitions. These icons are not necessary for constructing activity diagrams, but many users prefer the added impact that they provide.

3.89.1 Notation

3.89.1.1 Signal receipt

The receipt of a signal may be shown as a concave pentagon that looks like a rectangle with a triangular notch in its side (either side). The signature of the signal is shown inside the symbol. An unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. A dashed arrow may be drawn from an object symbol to the notch on the pentagon to show the sender of the signal; this is optional.

3.89.1.2 Signal sending

The sending of a signal may be shown as a convex pentagon that looks like a rectangle with a triangular point on one side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. A dashed arrow may be drawn from the point on the pentagon to an object symbol to show the receiver of the signal, this is optional.

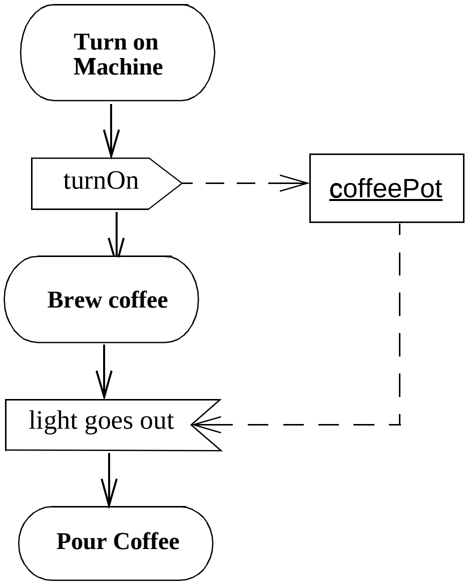

Figure 3-78 Symbols for Signal Receipt and Sending

3.89.1.3 *Deferred events*

A frequent situation is when an event that occurs must be “deferred” for later use while some other action or subactivity is underway. (Normally an event that is not handled immediately is lost.) This may be thought of as having an internal transition that handles the event and places it on an internal queue until it is needed or until it is discarded. Each state specifies a set of events that are deferred if they occur during the state and are not used to trigger a transition. If an event is not included in the set of deferrable events for a state, and it does not trigger a transition, then it is discarded from the queue even if it has already occurred. If a transition depends on an event, the transition fires immediately if the event is already on the internal queue. If several transitions are possible, the leading event in the queue takes precedence.

A deferrable event is shown by listing it within the state followed by a slash and the special operation *defer.* If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again. When the object reaches a state in which the event is not deferred, it must be accepted or lost. The indication may be placed on a composite state or its equivalents, submachine and subactivity states, in which case it remains deferrable throughout the composite state. A contained transition may still be triggered by a deferrable event, whereupon it is removed from the queue.

It is not necessary to defer events on action states, because these states are not interruptible for event processing. In this case, both deferred and undeferred events that occur during the state are deferred until the state is completed. This means that the timing of the transition will be the same regardless of the relative order of the event and the state completion, and regardless of whether events are deferred.

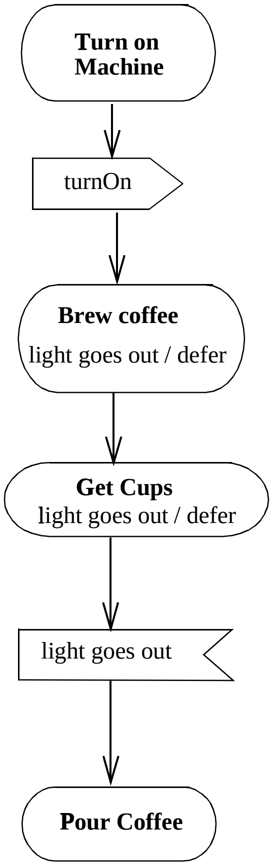

Figure 3-79 Deferred Event

3.89.2 Mapping

A signal receipt symbol maps into a state with no actions or internal transitions. Its specified event maps to a trigger event on the outgoing transition between it and the following state.

A signal send symbol maps into a SendAction on the incoming transition between it and the previous state.

A deferred event attached to a state maps into a *deferredEvent* association from the State to the Event.

3.90 Synch States

The SynchState notation may be omitted in Activity Diagrams when a SynchState has one incoming and one outgoing transition, and an unlimited bound. The semantics and mapping are the same as if the synch state circles were included, as defined for state machine notation.

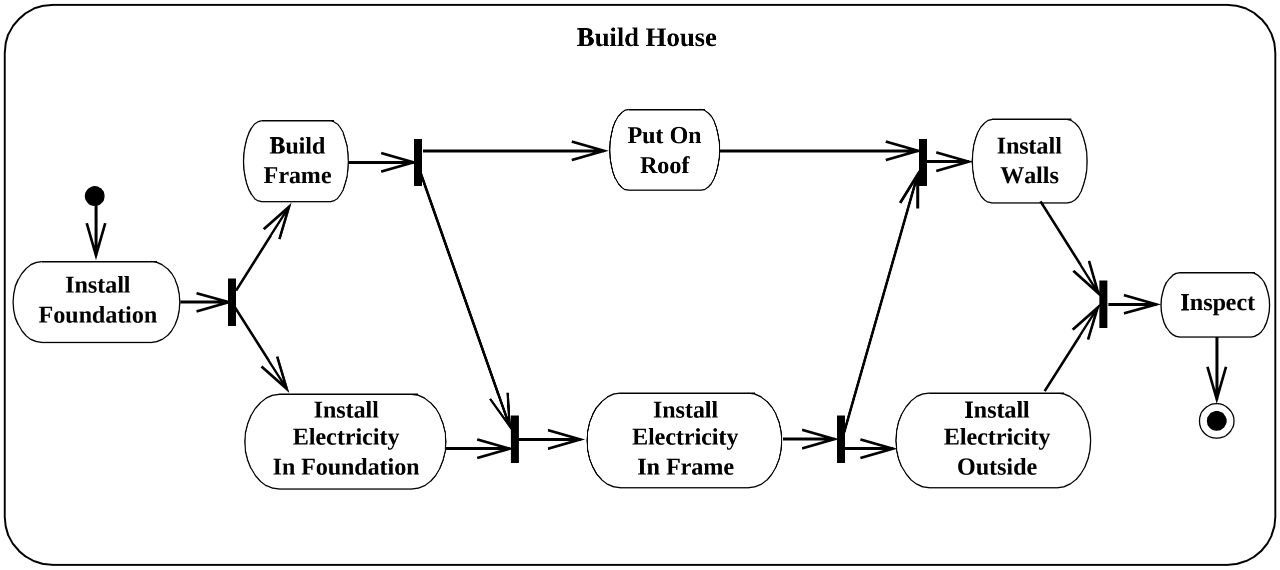

Figure 3-80 Synchronizing parallel activities

3.91 Dynamic Invocation

3.91.1 Semantics

The actions of an action state or the activity graph of a subactivity state may be executed more than once concurrently. The number of concurrent invocations is determined at runtime by a concurrency expression, which evaluates to a set of argument lists, one argument list for each invocation.

3.91.2 Notation

If the dynamic concurrency of an action or subactivity state is not always exactly one, its multiplicity is shown in the upper right corner of the state. Otherwise, nothing is shown.

3.91.3 Mapping

A multiplicity string in the upper right corner of an action or subactivity state maps to the same value in the dynamicMultiplicity attribute of the state. The presence of a multiplicity string also maps to a value of true for the isDynamic attribute of the state. If no multiplicity is present, the value of the isDynamic attribute is false.

3.92 Conditional Forks

In Activity Diagrams, transitions outgoing from forks may have guards. This means the region initiated by a fork transition might not start, and therefore is not required to complete at the corresponding join. The usual notation and mapping for guards may be used on the transition outgoing from a fork.

Part 11 - Implementation Diagrams

Implementation diagrams show aspects of implementation, including source code structure and run-time implementation structure. They come in two forms:

1. component diagrams show the structure of the code itself and
2. deployment diagrams show the structure of the run-time system.

They can also be applied in a broader sense to business modeling in which the “code” components are the business procedures and documents and the “run-time structure” is the organization units and resources (human and other) of the business.

3.93 Component Diagram

3.93.1 Semantics

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. For a business, “software” components are taken in the broad sense to include business procedures and documents. A software module may be represented as a component stereotype. Some components exist at compile time, some exist at link time, some exist at run time, and some exist at more than one time. A compile-only component is one that is only meaningful at compile time. The run-time component in this case would be an executable program.

A component diagram has only a type form, not an instance form. To show component instances, use a deployment diagram (possibly a degenerate one without nodes).

3.93.2 Notation

A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.