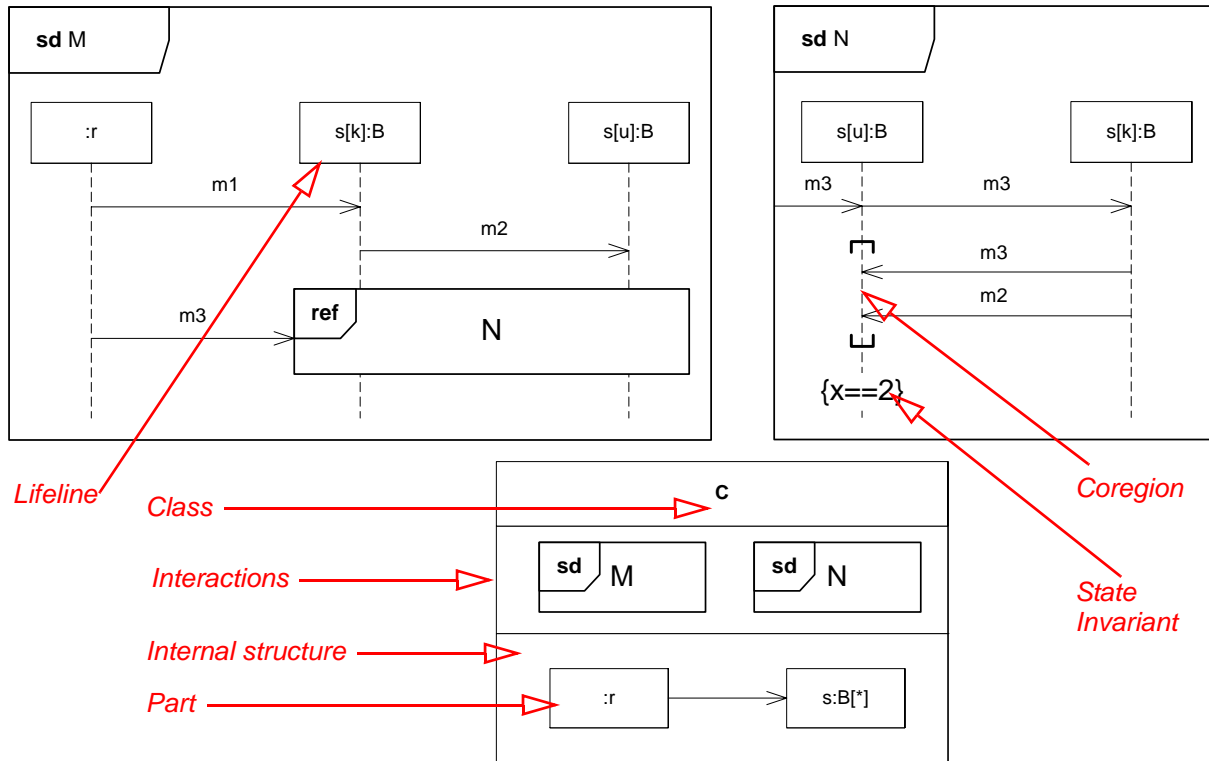


## Examples



**Figure 14.22 - Sequence Diagrams where two Lifelines refer to the same set of Parts (and Internal Structure)**

The sequence diagrams shown in Figure 14.22 show a scenario where `r` sends `m1` to `s[k]` (which is of type `B`), and `s[k]` sends `m2` to `s[u]`. In the meantime independent of `s[k]` and `s[u]`, `r` may have sent `m3` towards the **InteractionUse** `N` through a gate. Following the `m3` message into `N` we see that `s[u]` then sends another `m3` message to `s[k]`. `s[k]` then sends `m3` and then `m2` towards `s[u]`. `s[u]` receives the two latter messages in any order (coregion). Having received these messages, we state an invariant on a variable `x` (most certainly owned by `s[u]`).

In order to explain the mapping of the notation onto the metamodel we have pointed out areas and their corresponding metamodel concept in Figure 14.23. Let us go through the simple diagram and explain how the metamodel is built up. The whole diagram is an **Interaction** (named `N`). There is a formal gate (with implicit name `in_m3`) and two **Lifelines** (named `s[u]` and `s[k]`) that are contained in the **Interaction**. Furthermore the two **Messages** (occurrences) both of the same type `m3`, implicitly named `m3_1` and `m3_2` here, are also owned by the **Interaction**. Finally there are the three **OccurrenceSpecifications**.

We have omitted in this metamodel the objects that are more peripheral to the **Interaction** model, such as the **Part** `s` and the class `B` and the connector referred by the **Message**.

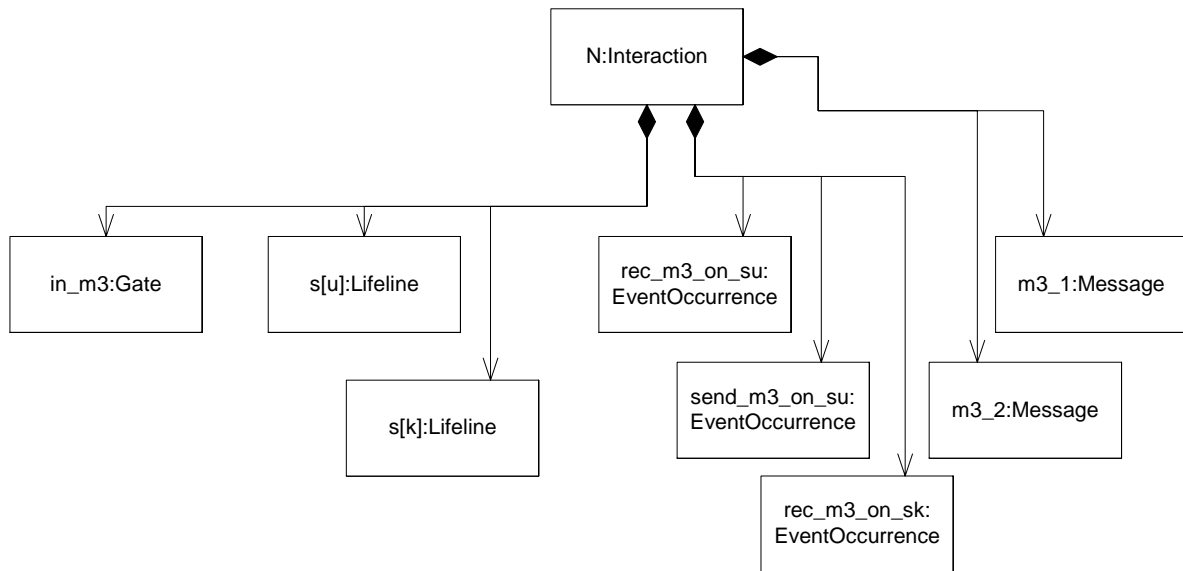
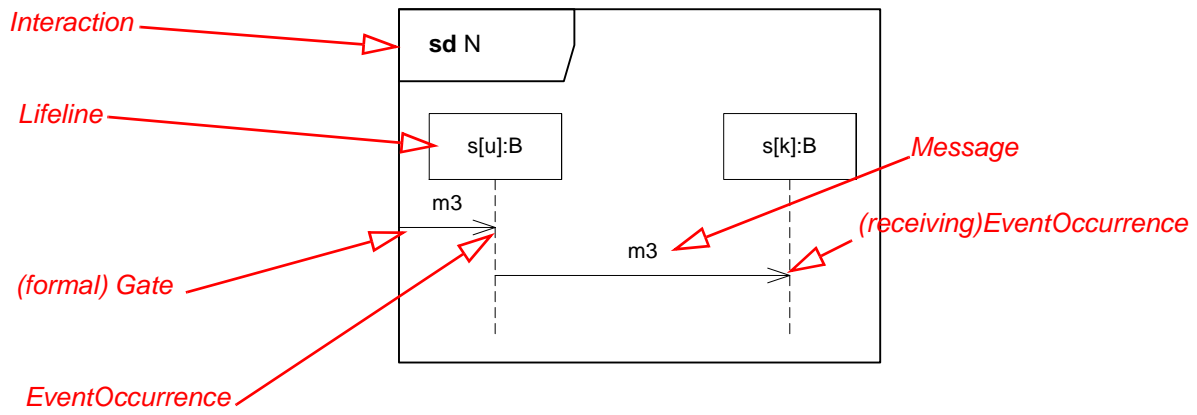
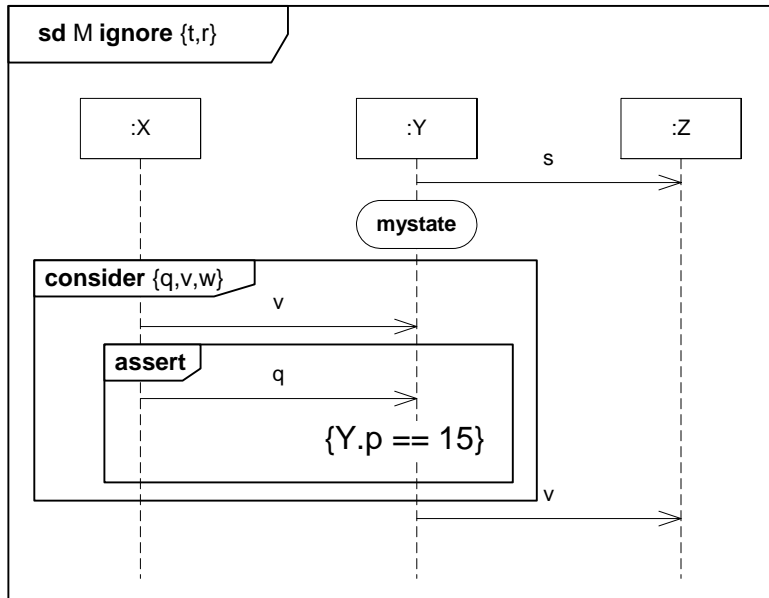


Figure 14.23 - Metamodel elements of a sequence diagram



**Figure 14.24 - Ignore, Consider, assert with State Invariants**

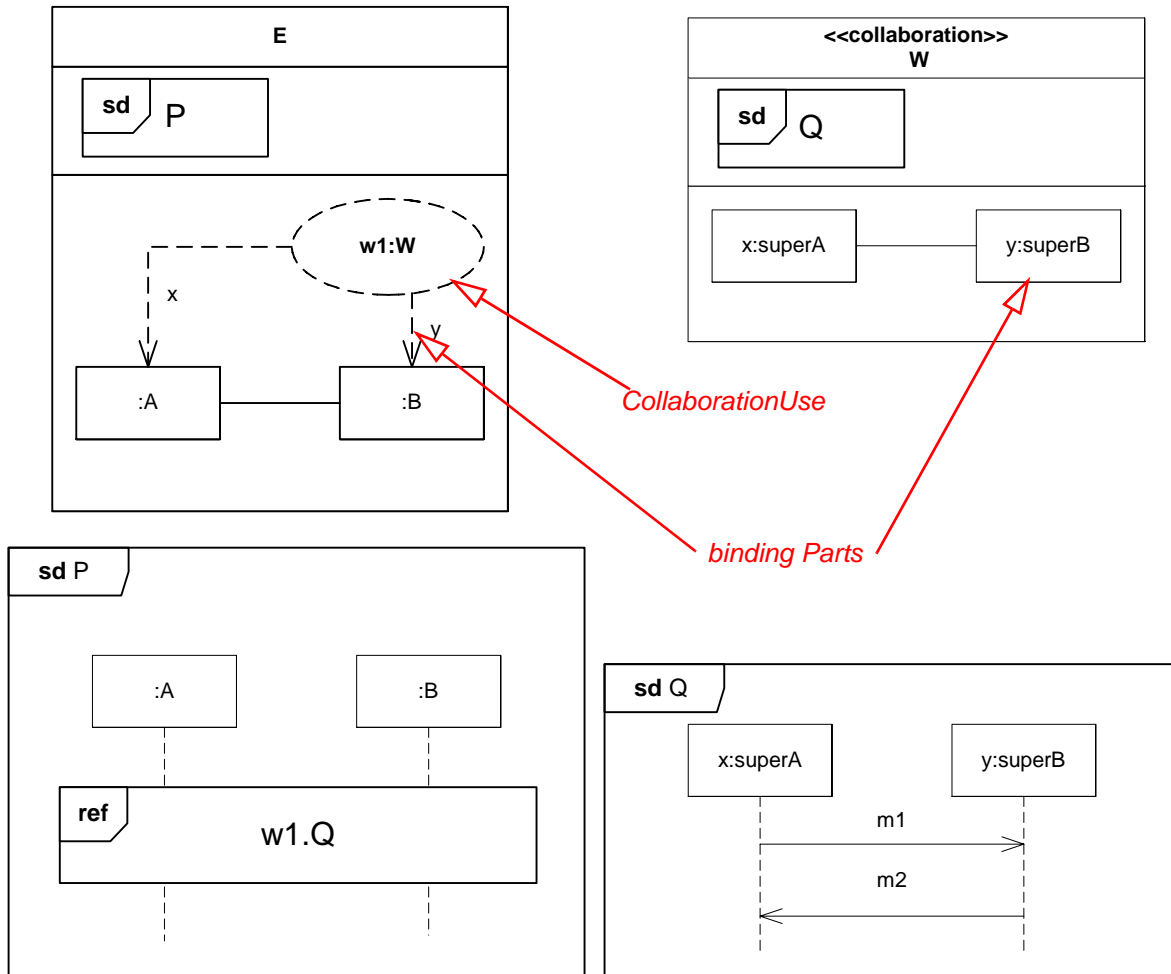
In Figure 14.24 we have an Interaction M, which considers message types other than t and r. This means that if this Interaction is used to specify a test of an existing system and when running that system a t or an r occurs, these messages will be ignored by this specification. t and r will of course be handled in some manner by the running system, but how they are handled is irrelevant for our Interaction shown here.

The State invariant given as a state “mystate” will be evaluated at runtime directly prior to whatever event occurs on Y after “mystate.” This may be the reception of q as specified within the assert-fragment, or it may be an event that is specified to be insignificant by the filters.

The **assert** fragment is nested in a **consider** fragment to mean that we expect a q message to occur once a v has occurred here. Any occurrences of messages other than v, w, and q will be ignored in a test situation. Thus the appearance of a w message after the v is an invalid trace.

The state invariant given in curly brackets will be evaluated prior to the next event occurrence after that on Y.

## Internal Structure and corresponding Collaboration Uses

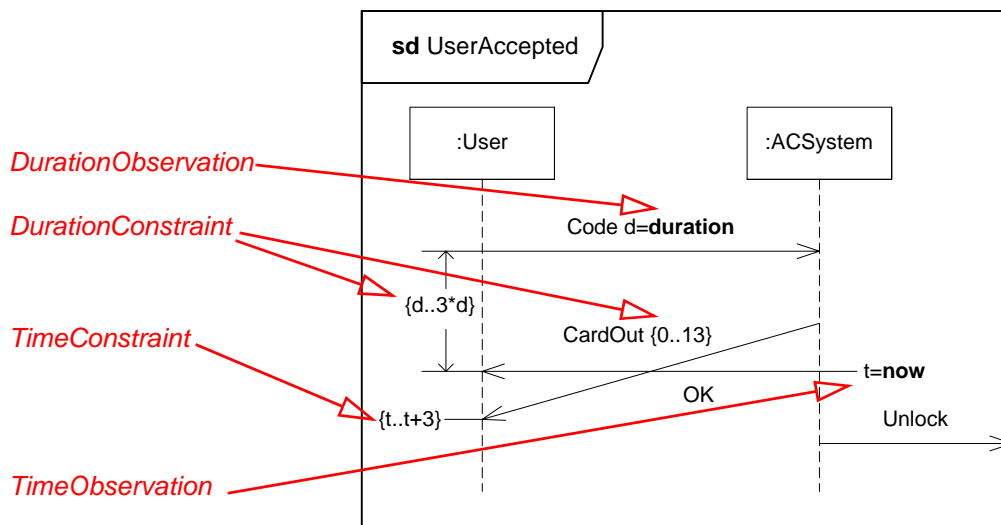


**Figure 14.25 - Describing collaborations and their binding**

The example in Figure 14.25 shows how collaboration uses are employed to make Interactions of a Collaboration available in another classifier.

The collaboration **W** has two parts **x** and **y** that are of types (classes) **superA** and **superB** respectively. Classes **A** and **B** are specializations of **superA** and **superB** respectively. The Sequence Diagram **Q** shows a simple Interaction that we will reuse in another environment. The class **E** represents this other environment. There are two anonymous parts **:A** and **:B** and the CollaborationUse **w1** of Collaboration **W** binds **x** and **y** to **:A** and **:B** respectively. This binding is legal since **:A** and **:B** are parts of types that are specializations of the types of **x** and **y**.

In the Sequence Diagram **P** (owned by class **E**) we use the Interaction **Q** made available via the CollaborationUse **w1**.



**Figure 14.26 - Sequence Diagram with time and timing concepts**

The Sequence Diagram in Figure 14.26 shows how time and timing notation may be applied to describe time observation and timing constraints. The `:User` sends a message `Code` and its duration is measured. The `:ACSystem` will send two messages back to the `:User`. `CardOut` is constrained to last between 0 and 13 time units. Furthermore the interval between the sending of `Code` and the reception of `OK` is constrained to last between  $d$  and  $3*d$  where  $d$  is the measured duration of the `Code` signal. We also notice the observation of the time point  $t$  at the sending of `OK` and how this is used to constrain the time point of the reception of `CardOut`.

## Communication Diagrams

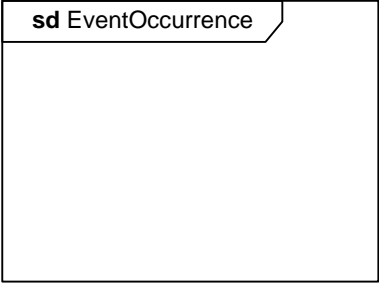
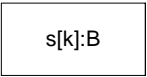
Communication Diagrams focus on the interaction between Lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of Messages is given through a sequence numbering scheme.

Communication Diagrams correspond to simple Sequence Diagrams that use none of the structuring mechanisms such as `InteractionUses` and `CombinedFragments`. It is also assumed that message overtaking (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.

## Graphical Nodes

Communication diagram nodes are shown in Table 14.3.

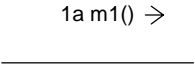
**Table 14.3 - Graphic nodes included in communication diagrams**

Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 467.
Lifeline		See “Lifeline (from BasicInteractions, Fragments)” on page 475.

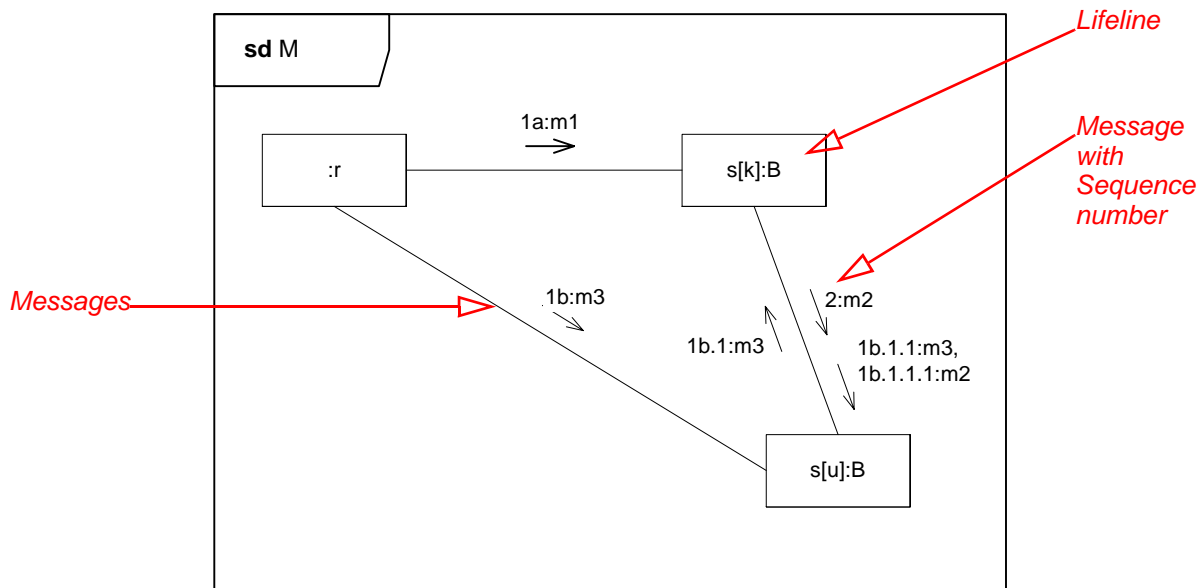
## Graphic Paths

Graphic paths of communication diagrams are given in Table 14.4.

**Table 14.4 - Graphic paths included in communication diagrams**

Node Type	Notation	Reference
Message		See “Message (from BasicInteractions)” on page 477 and “Sequence expression” on page 498. The arrow shown here indicates the communication direction.

## Examples



**Figure 14.27 - Communication diagram**

The Interaction described by a Communication Diagram in Figure 14.27 shows messages m1 and m3 being sent concurrently from :r towards two instances of the part s. The sequence numbers show how the other messages are sequenced. 1b.1 follows after 1b and 1b.1.1 thereafter etc. 2 follows after 1a and 1b.

### Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (':').

*sequence-term* '.' ... ':'

Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[ *integer* | *name* ] [ *recurrence* ]

The *integer* represents the sequential order of the Message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows Message 3.1.3 within activation 3.1. The *name* represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: Message 3.1a and Message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more Messages that are executed depending on the conditions involved. The choices are:

'\*' '[' *iteration-clause* ']' *an iteration*  
 '[' *guard* ']' *a branch*

An iteration represents a sequence of Messages at the given nesting depth. The iteration clause may be omitted (in which case the iteration conditions are unspecified). The iteration-clause is meant to be expressed in pseudocode or an actual programming language, UML does not prescribe its format. An example would be: *\*[i := 1..n]*.

A guard represents a Message whose execution is contingent on the truth of the condition clause. The guard is meant to be expressed in pseudocode or an actual programming language; UML does not prescribe its format. An example would be: *[x > y]*.

Note that a branch is notated the same as an iteration without a star. One might think of it as an iteration restricted to a single occurrence.

The iteration notation assumes that the Messages in the iteration will be executed sequentially. There is also the possibility of executing them concurrently. The notation for this is to follow the star by a double vertical line (for parallelism): *\*//*.

Note that in a nested control structure, the recurrence is not repeated at inner levels. Each level of structure specifies its own iteration within the enclosing context.

## Interaction Overview Diagrams

Interaction Overview Diagrams define Interactions (described in Chapter 14, “Interactions”) through a variant of Activity Diagrams (described in Chapter 6, “Activities”) in a way that promotes overview of the control flow.

Interaction Overview Diagrams focus on the overview of the flow of control where the nodes are Interactions or InteractionUses. The Lifelines and the Messages do not appear at this overview level.

## Graphic Nodes

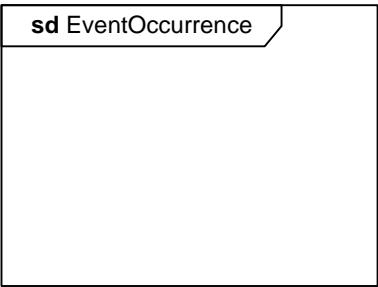
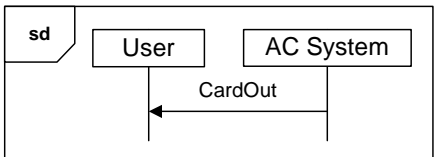
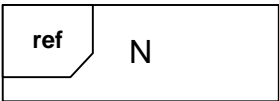
Interaction Overview Diagrams are specialization of Activity Diagrams that represent Interactions.

Interaction Overview Diagrams differ from Activity Diagrams in some respects.

1. In place of ObjectNodes of Activity Diagrams, Interaction Overview Diagrams can only have either (inline) Interactions or InteractionUses. Inline Interaction diagrams and InteractionUses are considered special forms of ActivityInvocations.
2. Alternative Combined Fragments are represented by a Decision Node and a corresponding Merge Node.
3. Parallel Combined Fragments are represented by a Fork Node and a corresponding Join Node.
4. Loop Combined Fragments are represented by simple cycles.
5. Branching and joining of branches must in Interaction Overview Diagrams be properly nested. This is more restrictive than in Activity Diagrams.
6. Interaction Overview Diagrams are framed by the same kind of frame that encloses other forms of Interaction Diagrams. The heading text may also include a list of the contained Lifelines (that do not appear graphically).



**Table 14.5 - Graphic nodes included in Interaction Overview Diagrams in addition to those borrowed from Activity Diagrams**

Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 467.
Interaction		An Interaction diagram of any kind may appear inline as an ActivityInvocation. See “Interaction (from BasicInteraction, Fragments)” on page 467. The inline Interaction diagrams may be either anonymous (as here) or named.
InteractionUse		ActivityInvocation in the form of InteractionUse. See “InteractionUse (from Fragments)” on page 473. The tools may choose to “explode” the view of an InteractionUse into an inline Interaction with the name of the Interaction referred by the occurrence. The inline Interaction will then replace the occurrence by a replica of the definition Interaction where arguments have replaced parameters.

## Examples

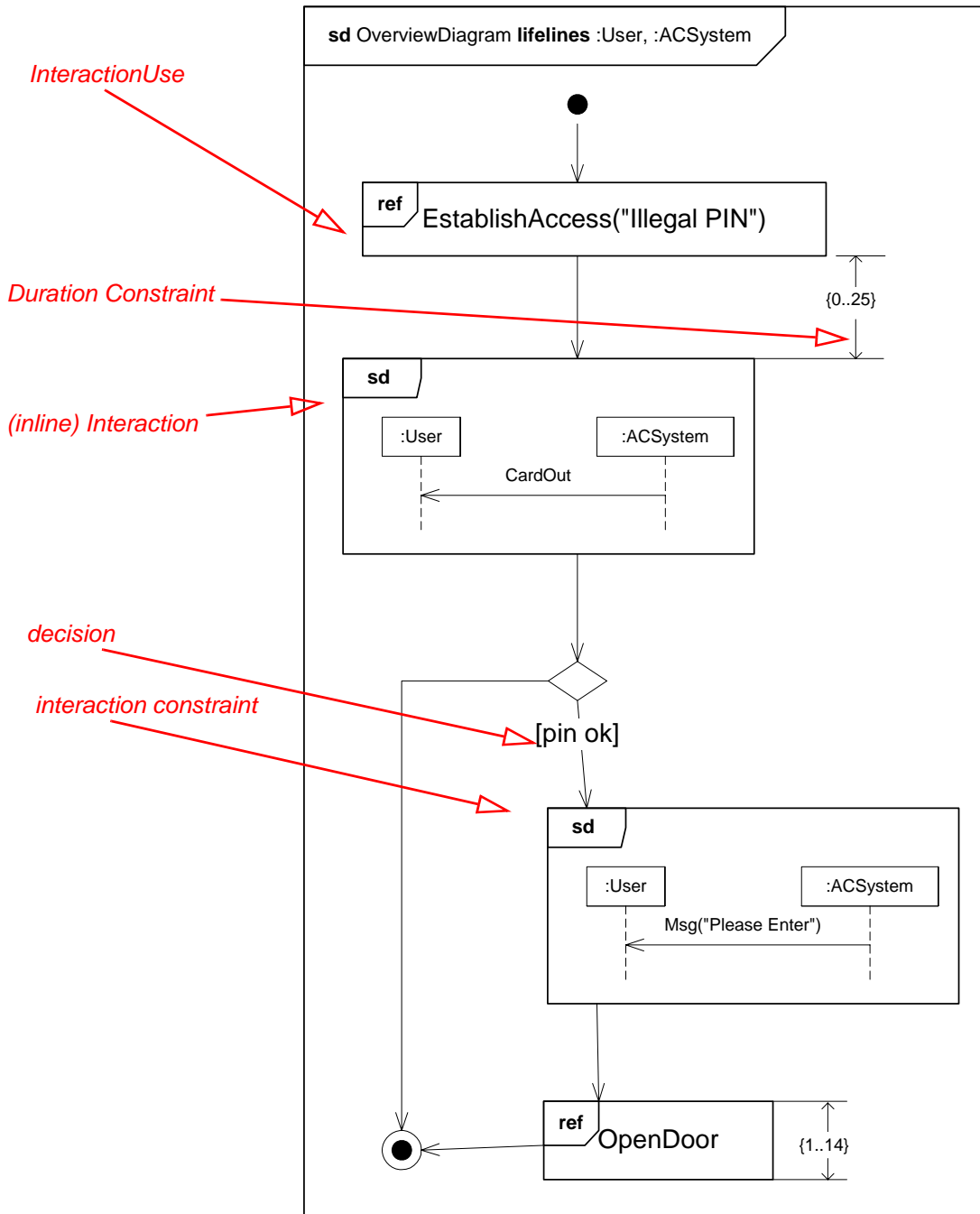


Figure 14.28 - Interaction Overview Diagram representing a High Level Interaction diagram

Interaction Overview Diagrams use Activity diagram notation where the nodes are either Interactions or InteractionUses. Interaction Overview Diagrams are a way to describe Interactions where Messages and Lifelines are abstracted away. In the purest form all Activities are InteractionUses and then there are no Messages or Lifelines shown in the diagram at all.

Figure 14.28 is another way to describe the behavior shown in Figure 14.17, with some added timing constraints. The Interaction *EstablishAccess* occurs first (with argument “Illegal PIN”) followed by weak sequencing with the message *CardOut* which is shown in an inline Interaction. Then there is an alternative as we find a decision node with an InteractionConstraint on one of the branches. Along that control flow we find another inline Interaction and an InteractionUse in (weak) sequence.

## Timing Diagram

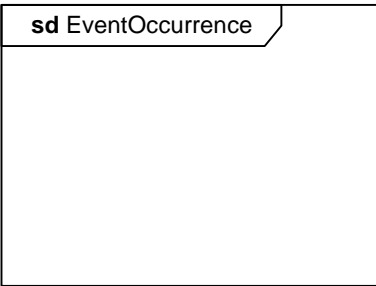
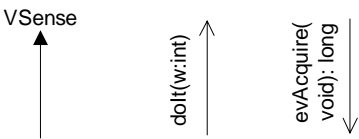
Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. Timing diagrams focus on conditions changing within and among Lifelines along a linear time axis.

Timing diagrams describe behavior of both individual classifiers and interactions of classifiers, focusing attention on time of occurrence of events causing changes in the modeled conditions of the Lifelines.

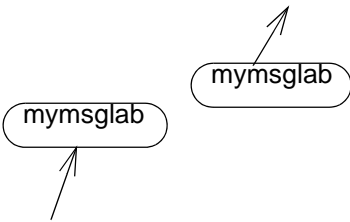
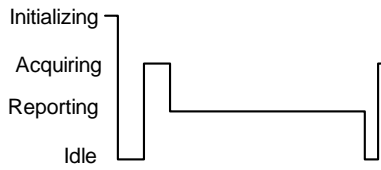

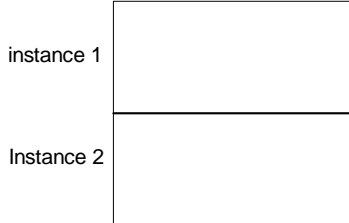


## Graphic Nodes

The graphic nodes and paths that can be included in timing diagrams are shown in Table 14.6.

**Table 14.6 - Graphic nodes and paths included in timing diagrams**

Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 467.
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. See “Message (from BasicInteractions)” on page 477.

**Table 14.6 - Graphic nodes and paths included in timing diagrams**

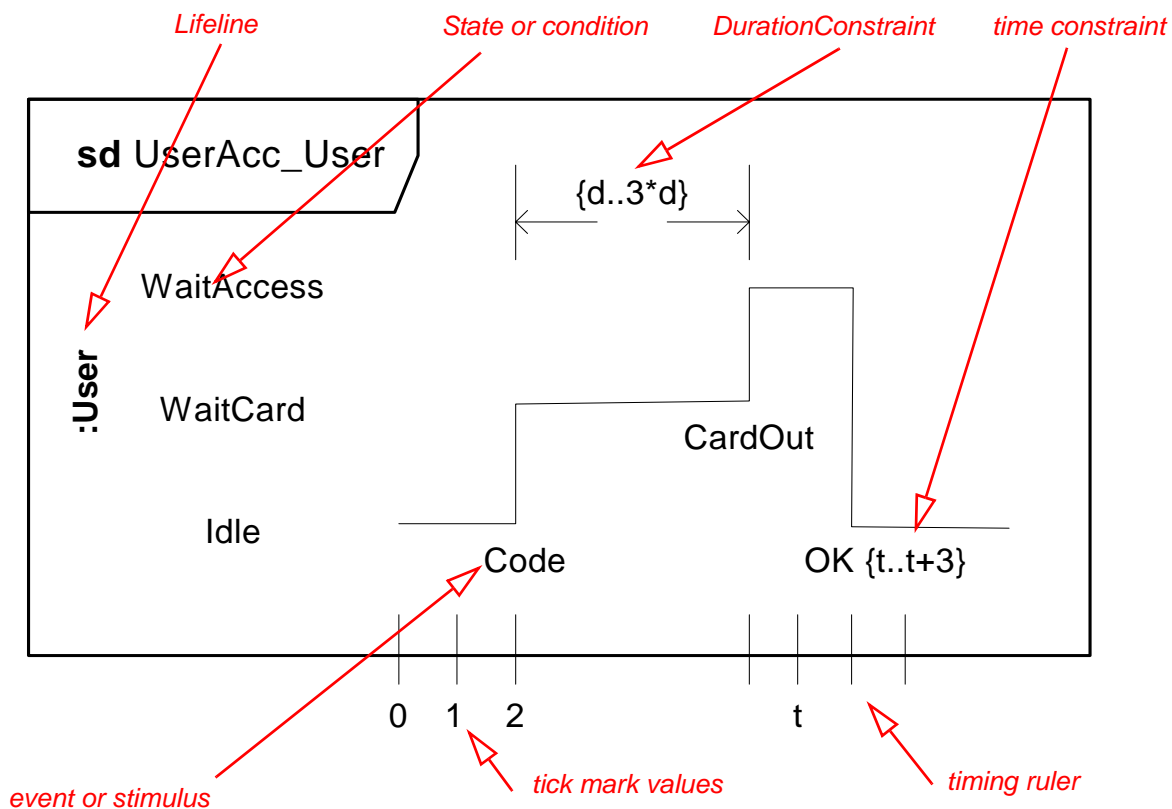
Node Type	Notation	Reference
Message label		Labels are only notational shorthands used to prevent cluttering of the diagrams with a number of messages crisscrossing the diagram between Lifelines that are far apart. The labels denote that a Message may be disrupted by introducing labels with the same name.
State or condition timeline		<p>This is the state of the classifier or attribute, or some testable condition, such as an discrete enumerable value. See also “StateInvariant (from BasicInteractions)” on page 487.</p> <p>It is also permissible to let the state-dimension be continuous as well as discrete. This is illustrative for scenarios where certain entities undergo continuous state changes, such as temperature or density.</p>
General value lifeline		Shows the value of the connectable element as a function of time. Value is explicitly denoted as text. Crossing reflects the event where the value changed.
Lifeline		See “Lifeline (from BasicInteractions, Fragments)” on page 475.
GeneralOrdering		See “GeneralOrdering (from BasicInteractions)” on page 466
DestructionEvent		See “DestructionEvent (from BasicInteractions)” on page 462.

## Examples

Timing diagrams show change in state or other condition of a structural element over time. There are a few forms in use. We shall give examples of the simplest forms.

Sequence Diagrams as the primary form of Interactions may also depict time observation and timing constraints. We show in Figure 14.26 an example in Sequence Diagram that we will also give in Timing Diagrams.

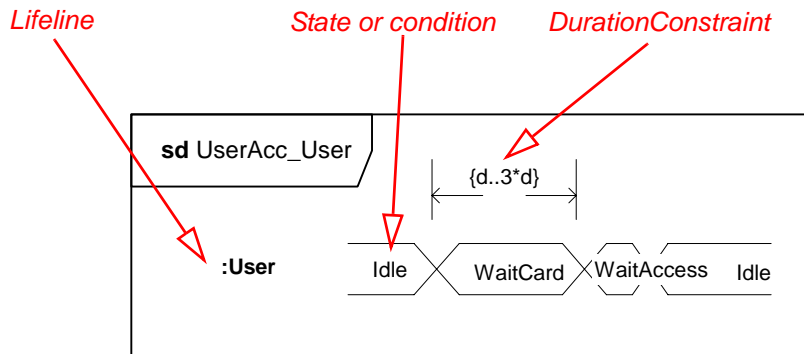
The :User of the Sequence Diagram in Figure 14.26 is depicted with a simple Timing Diagram in Figure 14.29.



**Figure 14.29 - A Lifeline for a discrete object**

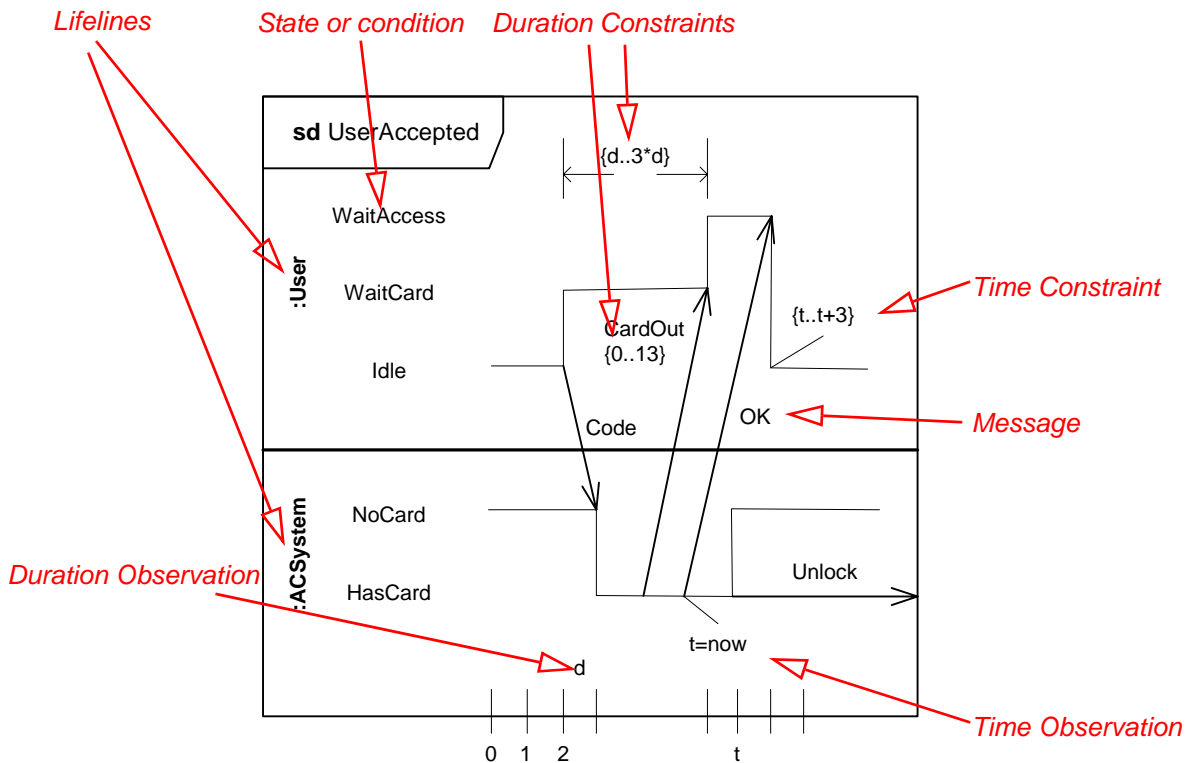
The primary purpose of the timing diagram is to show the change in state or condition of a lifeline (representing a Classifier Instance or Classifier Role) over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli. The received events are annotated as shown when it is desirable to show the event causing the change in condition or state.

Sometimes it is more economical and compact to show the state or condition on the vertical Lifeline as shown in Figure 14.30.



**Figure 14.30 - Compact Lifeline with States**

Finally we may have an elaborate form of TimingDiagrams where more than one Lifeline is shown and where the messages are also depicted. We show such a Timing Diagram in Figure 14.31 corresponding to the Sequence Diagram in Figure 14.26.



**Figure 14.31 - Timing Diagram with more than one Lifeline and with Messages**

### Changes from previous UML

The Timing Diagrams were not available in UML 1.4.



## 15 State Machines

### 15.1 Overview

The StateMachine package defines a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system. These two kinds of state machines are referred to here as *behavioral state machines* and *protocol state machines*.

#### **Behavioral state machines**

State machines can be used to specify behavior of various model elements. For example, they can be used to model the behavior of individual entities (e.g., class instances). The state machine formalism described in this section is an object-based variant of Harel statecharts.

#### **Protocol State machines**

Protocol state machines are used to express usage protocols. Protocol state machines express the legal transitions that a classifier can trigger. The state machine notation is a convenient way to define a lifecycle for objects, or an order of the invocation of its operation. Because protocol state machines do not preclude any specific behavioral implementation, and enforces legal usage scenarios of classifiers, interfaces, and ports can be associated to this kind of state machines.



## 15.2 Abstract Syntax

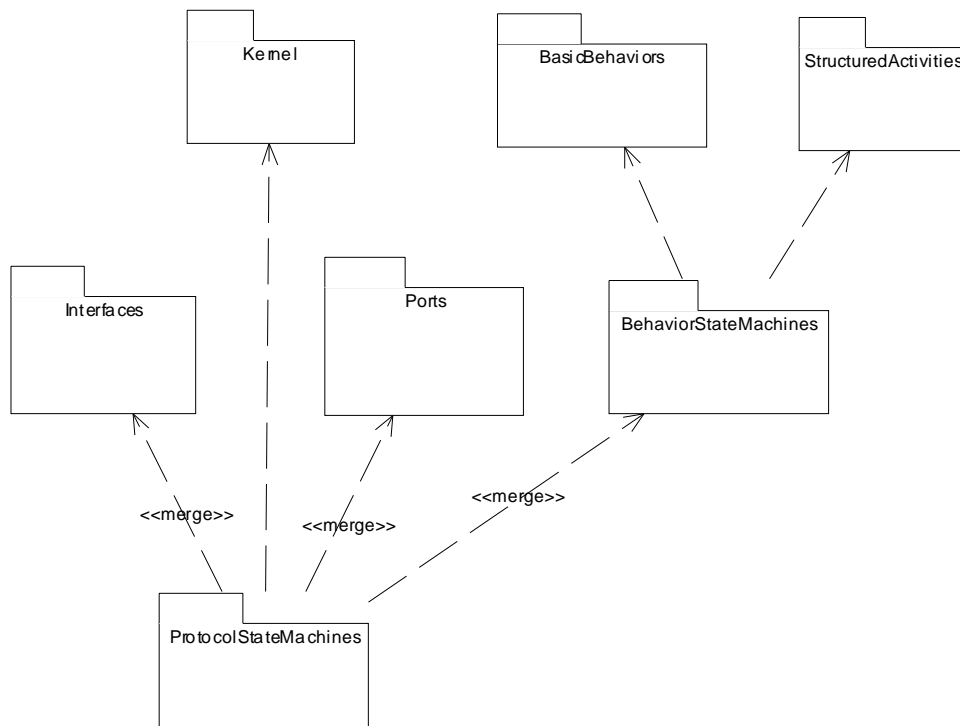


Figure 15.1 - Package Dependencies

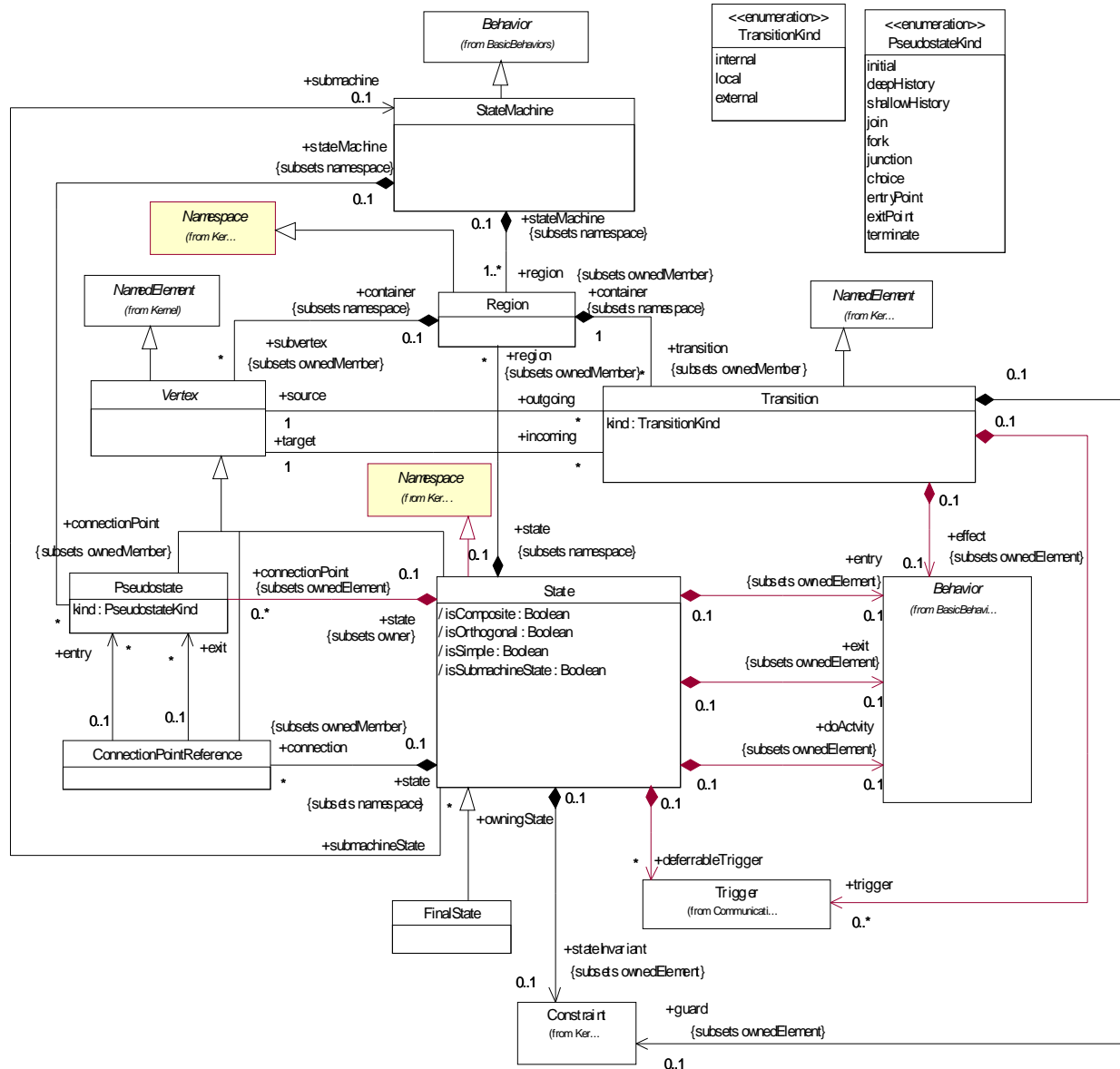


Figure 15.2 - State Machines

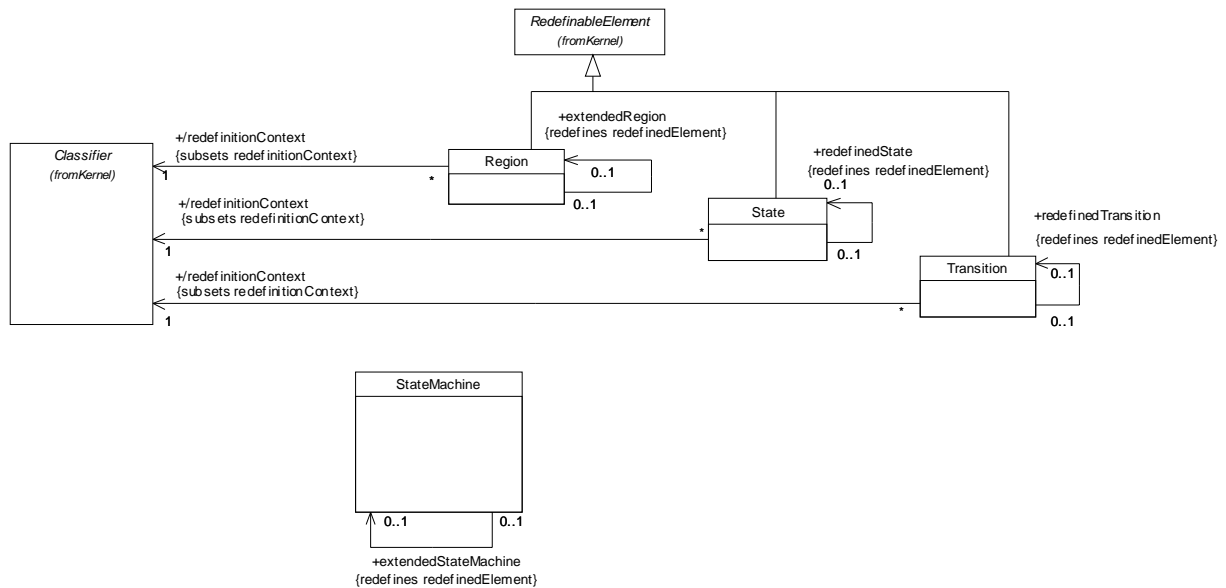


Figure 15.3 - State Machine Redefinitions

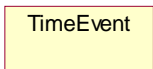


Figure 15.4 - Time events

Package *ProtocolStateMachines*

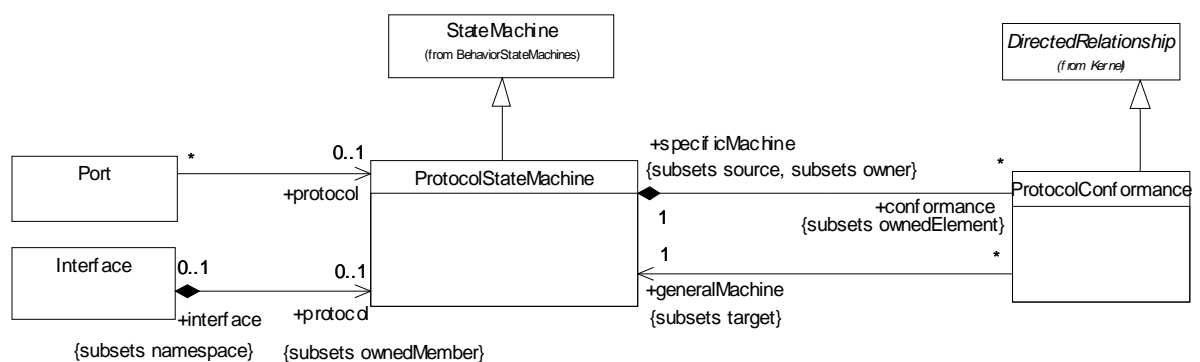


Figure 15.5 - Protocol State Machines

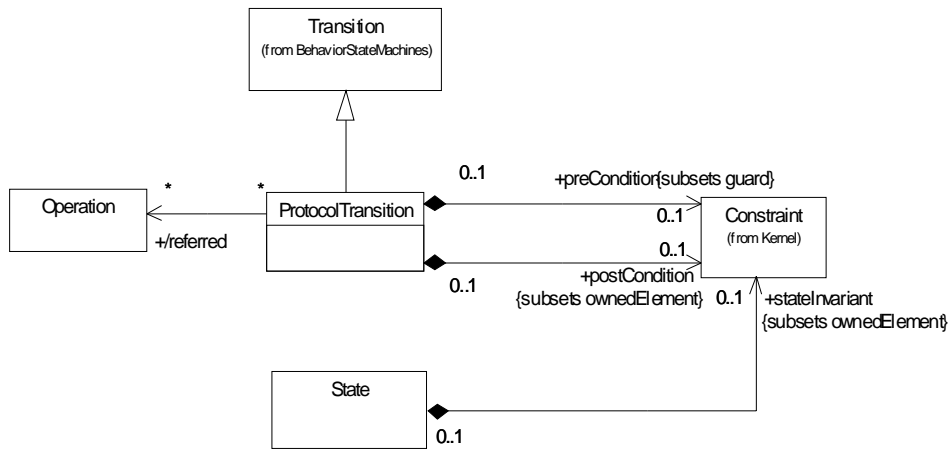


Figure 15.6 - Constraints

## 15.3 Class Descriptions

### 15.3.1 ConnectionPointReference (from BehaviorStateMachines)

A connection point reference represents a usage (as part of a submachine state) of an entry/exit point defined in the statemachine reference by the submachine state.

#### Generalizations

- “Vertex (from BehaviorStateMachines)” on page 562

#### Description

Connection point references of a submachine state can be used as sources/targets of transitions. They represent entries into or exits out of the submachine state machine referenced by the submachine state.

#### Attributes

No additional attributes

#### Associations

- entry: Pseudostate[1..\*] The entryPoint kind pseudo states corresponding to this connection point.
- exit: Pseudostate[1..\*] The exitPoints kind pseudo states corresponding to this connection point.
- state : State [0..1] The State in which the connection point refreshens are defined.  
{Subsets *Element::namespace*}

#### Constraints

- [1] The entry Pseudostates must be Pseudostates with kind entryPoint.  
entry->notEmpty() implies entry->forall(e | e.kind = #entryPoint)

[2] The exit Pseudostates must be Pseudostates with kind exitPoint.  
exit->notEmpty() implies exit->forAll(e | e.kind = #exitPoint)

## Semantics

Connection point references are sources/targets of transitions implying exits out of/entries into the submachine state machine referenced by a submachine state.

An entry point connection point reference as the target of a transition implies that the target of the transition is the entry point pseudostate as defined in the submachine of the submachine state. As a result, the regions of the submachine state machine are entered at the corresponding entry point pseudo states.

An exit point connection point reference as the source of a transition implies that the source of the transition is the exit point pseudostate as defined in the submachine of the submachine state that has the exit point connection point defined. When a region of the submachine state machine has reached the corresponding exit points, the submachine state exits at this exit point.

## Notation

A connection point reference to an entry point has the same notation as an entry point pseudostate. The circle is placed on the border of the state symbol of a submachine state.

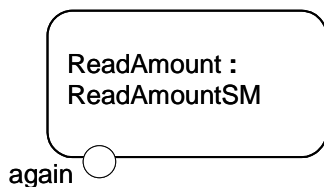


Figure 15.7 - Entry Point

A connection point reference to an exit point has the same notation as an exit point pseudostate. The encircled cross is placed on the border of the state symbol of a submachine state.

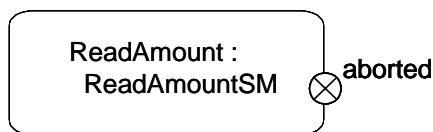
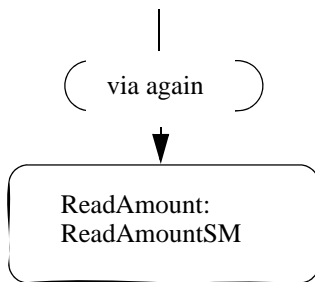


Figure 15.8 - Exit Point

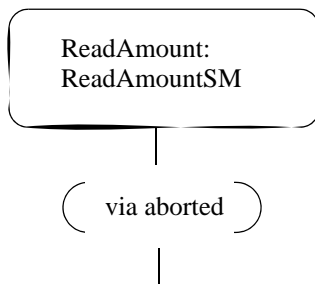
## Presentation Options

A connection point reference to an entry point can also be visualized using a rectangular symbol as shown in Figure 15.9. The text inside the symbol shall contain the keyword 'via' followed by the name of the connection point. This notation may only be used if the transition ending with the connection point is defined using the transition-oriented control icon notation as defined in "Transition (from BehaviorStateMachines)" on page 553.



**Figure 15.9 - Alternative Entry Point notation**

A connection point reference to an exit point can also be visualized using a rectangular symbol as shown in Figure 15.9. The text inside the symbol shall contain the keyword ‘via’ followed by the name of the connection point. This notation may only be used if the transition associated with the connection point is defined using the transition-oriented control icon notation as defined in “Transition (from BehaviorStateMachines)” on page 553.



**Figure 15.10 - Alternative Exit Point notation**

### 15.3.2 FinalState (from BehaviorStateMachines)

#### Generalizations

- “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 531

#### Description

A special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed.

#### Attributes

No additional attributes

#### Associations

No additional associations

## Constraints

- [1] A final state cannot have any outgoing transitions.  
`self.outgoing->size() = 0`
- [2] A final state cannot have regions.
- [3] A final state cannot reference a submachine.
- [4] A final state has no entry behavior.
- [5] A final state has no exit behavior.
- [6] A final state has no state (doActivity) behavior.

## Semantics

When the final state is entered, its containing region is completed, which means that it satisfies the completion condition. The containing state for this region is considered completed when all contained regions are completed. If the region is contained in a state machine and all other regions in the state machine also are completed, the entire state machine terminates, implying the termination of the context object of the state machine.

## Notation

A final state is shown as a circle surrounding a small solid filled circle (see Figure 15.11). The corresponding completion transition on the enclosing state has as notation an unlabeled transition.



Figure 15.11 - Final State

## Example

Figure 15.33 on page 540 has an example of a final state (the right most of the states within the composite state).

## 15.3.3 Interface (from ProtocolStateMachines)

Interface is defined as a specialization of the general Interface, adding an association to a protocol state machine.

### Generalizations

- “Interface (from Communications)” on page 429 (*merge increment*)

### Description

Since an interface specifies conformance characteristics, it does not own detailed behavior specifications. Instead, interfaces may own a protocol state machine that specifies event sequences and pre/post conditions for the operations and receptions described by the interface.

### Attributes

No additional attributes

## Associations

- protocol: ProtocolStateMachine [0..1]      References a protocol state machine specifying the legal sequences of the invocation of the behavioral features described in the interface.

## Semantics

Interfaces can specify behavioral constraints on the features using a protocol state machine. A classifier realizing an interface must comply with the protocol state machine owned by the interface.

## Changes from previous UML

Interfaces can own a protocol state machine.

### 15.3.4 Port (from ProtocolStateMachines)

#### Generalizations

- “Port (from Ports)” on page 175 (*merge increment*)

#### Description

Port is defined as a specialization of the general Port, adding an association to a protocol state machine.

#### Attributes

No additional attributes

#### Associations

- protocol: ProtocolStateMachine [0..1]      References an optional protocol state machine that describes valid interactions at this interaction point.

## Semantics

The protocol references a protocol state machine (see “ProtocolStateMachine (from ProtocolStateMachines)” on page 516) that describes valid sequences of operation and reception invocations that may occur at this port.

### 15.3.5 ProtocolConformance (from ProtocolStateMachines)

#### Generalizations

- “DirectedRelationship (from Kernel)” on page 59

#### Description

Protocol state machines can be redefined into more specific protocol state machines, or into behavioral state machines. Protocol conformance declares that the specific protocol state machine specifies a protocol that conforms to the general state machine one, or that the specific behavioral state machine abide by the protocol of the general protocol state machine.

A protocol state machine is owned by a classifier. The classifiers owning a general state machine and an associated specific state machine are generally also connected by a generalization or a realization link.



## Attributes

No additional attributes

## Associations

- `specificMachine: ProtocolStateMachine [1]` Specifies the state machine that conforms to the general state machine.
- `generalMachine: ProtocolStateMachine [1]` Specifies the protocol state machine to which the specific state machine conforms.

## Constraints

No additional constraints

## Semantics

Protocol conformance means that every rule and constraint specified for the general protocol state machine (state invariants, pre and post conditions for the operations referred by the protocol state machine) apply to the specific protocol or behavioral state machine.

In most cases there are relationships between the classifier being the context of the specific state machine and the classifier being the context of the general protocol state machine. Generally, the former specializes or realizes the latter. It is also possible that the specific state machine is a behavioral state machine that implements the general protocol state machine, both state machines having the same class as a context.

## 15.3.6 ProtocolStateMachine (from ProtocolStateMachines)

### Generalizations

- “StateMachine (from BehaviorStateMachines)” on page 545

### Description

A *protocol state machine* is always defined in the context of a classifier. It specifies which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifier’s operations. A protocol state machine presents the possible and permitted transitions on the instances of its context classifier, together with the operations that carry the transitions. In this manner, an instance lifecycle can be created for a classifier, by specifying the order in which the operations can be activated and the states through which an instance progresses during its existence.

## Attributes

No additional attributes

## Associations

- `conformance: ProtocolConformance[*]` Conformance between protocol state machines.

## Constraints

- [1] A protocol state machine must only have a classifier context, not a behavioral feature context.  
(**not** context->isEmpty( )) **and** specification->isEmpty()

- [2] All transitions of a protocol state machine must be protocol transitions. (transitions as extended by the ProtocolStateMachines package).
- ```
region->forAll(r | r.transition->forAll(t | t.ocllsTypeOf(ProtocolTransition)))
```
- [3] The states of a protocol state machine cannot have entry, exit, or do activity actions.
- ```
region->forAll(r | r.subvertex->forAll(v | v.ocllsKindOf(State) implies
(v.entry->isEmpty() and v.exit->isEmpty() and v.doActivity->isEmpty()))
```
- [4] Protocol state machines cannot have deep or shallow history pseudostates.
- ```
region->forAll (r | r.subvertex->forAll (v | v.ocllsKindOf(Pseudostate) implies
((v.kind <> #deepHistory) and (v.kind <> #shallowHistory))))
```
- [5] If two ports are connected, then the protocol state machine of the required interface (if defined) must be conformant to the protocol state machine of the provided interface (if defined).

## Semantics

Protocol state machines help define the usage mode of the operations and receptions of a classifier by specifying:

- In which context (under which states and pre conditions) they can be used.
- If there is a protocol order between them.
- What result is expected from their use.

The states of a protocol state machine (protocol states) present an external view of the class that is exposed to its clients. Depending on the context, protocol states can correspond to the internal states of the instances as expressed by behavioral state machines, or they can be different.

A protocol state machine expresses parts of the constraints that can be formulated for pre- and post conditions on operations. The translation from protocol state machine to pre- and post conditions on operations might not be straightforward, because the conditions would need to account for the operation call history on the instance, which may or may not be directly represented by its internal states. A protocol state machine provides a direct model of the state of interaction with the instance, so that constraints on interaction are more easily expressed.

The protocol state machine defines all allowed transitions for each operation. The protocol state machine must represent all operations that can generate a given change of state for a class. Those operations that do not generate a transition are not represented in the protocol state machine.

Protocol state machines constitute a means to formalize the interface of classes, and do not express anything except consistency rules for the implementation or dynamics of classes.

Protocol state machine interpretation can vary from:

1. Declarative protocol state machines that specify the legal transitions for each operation. The exact triggering condition for the operations is not specified. This specification only defines the contract for the user of the context classifier.
2. Executable protocol state machines, that specify all events that an object may receive and handle, together with the transitions that are implied. In this case, the legal transitions for operations will exactly be the triggered transitions. The call trigger specifies the effect action, which is the call of the associated operation.

The representation for both interpretations is the same, the only difference being the direct dynamic implication that the interpretation 2 provides.

Elaborated forms of state machine modeling such as compound transitions, sub statemachines, composite states, and concurrent regions can also be used for protocol state machines. For example, concurrent regions make it possible to express protocol where an instance can have several active states simultaneously. sub state machines and compound transitions are used as in behavioral state machines for factorizing complex protocol state machines.

A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having protocol state machine, when the protocols are orthogonal. An alternative to multiple protocol state machines can always be found by having one protocol state machine, with sub state machines in concurrent regions.

## Notation

The notation for protocol state machine is very similar to the one of behavioral state machines. The keyword {protocol} placed close to the name of the state machine differentiates graphically protocol state machine diagrams.

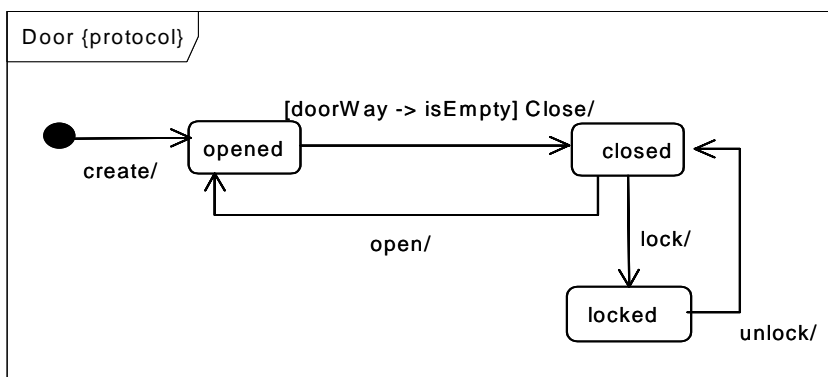


Figure 15.12 - Protocol state machine

## 15.3.7 ProtocolTransition (from ProtocolStateMachines)

### Generalizations

- “Transition (from BehaviorStateMachines)” on page 553

### Description

A protocol transition (transition as specialized in the ProtocolStateMachines package) specifies a legal transition for an operation. Transitions of protocol state machines have the following information: a pre condition (guard), on trigger, and a post condition. Every protocol transition is associated to zero or one operation (referred BehavioralFeature) that belongs to the context classifier of the protocol state machine.

The protocol transition specifies that the associated (referred) operation can be called for an instance in the origin state under the initial condition (guard), and that at the end of the transition, the destination state will be reached under the final condition (post).

### Attributes

No additional attributes

## Associations

- `\referred: Operation[0..*]` This association refers to the associated operation. It is derived from the operation of the call trigger when applicable.
- `postCondition: Constraint[0..1]` Specifies the post condition of the transition, which is the condition that should be obtained once the transition is triggered. This post condition is part of the post condition of the operation connected to the transition.
- `preCondition: Constraint[0..1]` Specifies the precondition of the transition. It specifies the condition that should be verified before triggering the transition. This guard condition added to the source state will be evaluated as part of the precondition of the operation referred by the transition if any.

## Constraints

- [1] A protocol transition always belongs to a protocol state machine.  
`container.belongsToPSM()`
- [2] A protocol transition never has associated actions.  
`effect->isEmpty()`
- [3] If a protocol transition refers to an operation (i.e., has a call trigger corresponding to an operation), then that operation should apply to the context classifier of the state machine of the protocol transition.

## Additional Operations

- [1] The operation `belongsToPSM ()` checks if the region belongs to a protocol state machine.

```
context Region::belongsToPSM () : Boolean
result = if not stateMachine->isEmpty() then
    oclIsTypeOf(ProtocolStateMachine)
else if not state->isEmpty() then
    state.container.belongsToPSM ()
else false
```

## Semantics

### *No “effect” action*

The effect action is never specified. It is implicit, when the transition has a call trigger: the effect action will be the operation specified by the call trigger. It is unspecified in the other cases, where the transition only defines that a given event can be received under a specific state and pre-condition, and that a transition will lead to another state under a specific post condition, whatever action will be made through this transition.

### *Unexpected event reception*

The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a semantic variation point: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behavior is defined in UML.

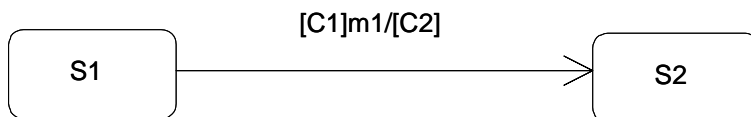
### *Unexpected behavior*

The interpretation of an unexpected behavior, that is an unexpected result of a transition (wrong final state or final state invariant, or post condition) is also a semantic variation point. However, this should be interpreted as an error of the implementation of the protocol state machine.

### *Equivalences to pre and post conditions of operations*

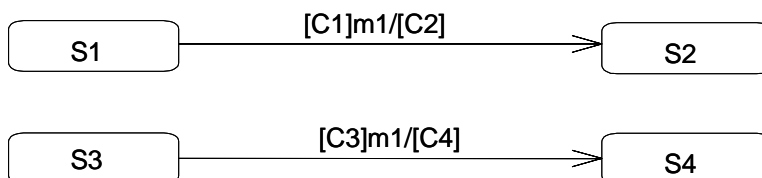
A protocol transition can be semantically interpreted in terms of pre- and post conditions on the associated operation. For example, the transition in Figure 15.13 can be interpreted in the following way:

1. The operation “m1” can be called on an instance when it is in the protocol state “S1” under the condition “C1.”
2. When “m1” is called in the protocol state “S1” under the condition “C1,” then the protocol state “S2” must be reached under the condition “C2.”



**Figure 15.13 - Example of a protocol transition associated to the "m1" operation**

### *Operations referred by several transitions*



**Figure 15.14 - Example of several transitions referring to the same operation**

In a protocol state machine, several transitions can refer to the same operation as illustrated below. In that case, all pre- and post conditions will be combined in the operation pre-condition as shown below:

Operation m1()

Pre: S1 is in the configuration state and C1

or

S3 is in the configuration state and C3

Post: if the initial condition was “S1 is in the configuration state and C1”

then S2 is in the configuration state and C2

else

if the initial condition was “S3 is in the configuration state and C3”

then S4 is in the configuration state and C4

A protocol state machine specifies all the legal transitions for each operation referred by its transitions. This means that for any operation referred by a protocol state machine, the part of its precondition relative to legal initial or final state is completely specified by the protocol state machine.

### *Unreferred Operations*

If an operation is not referred by any transition of a protocol state machine, then the operation can be called for any state of the protocol state machine, and does not change the current state.

### *Using events in protocol state machines*

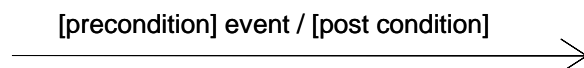
Apart from the operation call event, events are generally used for expressing a dynamic behavior interpretation of protocol state machines. An event that is not a call event can be specified on protocol transitions.

In this case, this specification is a requirement to the environment external to the state machine: it is legal to send this event to an instance of the context classifier only under the conditions specified by the protocol state machine.

Just like call event, this can also be interpreted in a dynamic way, as a semantic variation point.

### **Notation**

The usual state machine notation applies. The difference is that no actions are specified for protocol transitions, and that post conditions can exist. Post conditions have the same syntax as guard conditions, but appear at the end of the transition syntax.



**Figure 15.15 - Protocol transition notation**

### 15.3.8 Pseudostate (from BehaviorStateMachines)

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph.

#### Generalizations

- “Vertex (from BehaviorStateMachines)” on page 562

#### Description

Pseudostates are typically used to connect multiple transitions into more complex state transitions paths. For example, by combining a transition entering a fork pseudostate with a set of transitions exiting the fork pseudostate, we get a compound transition that leads to a set of orthogonal target states.

#### Attributes

- **kind**: PseudostateKind      Determines the precise type of the Pseudostate and can be one of: *entryPoint*, *exitPoint*, *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction*, *terminate* or *choice*.

#### Associations

- **stateMachine** : Statemachine [0..1]      The StateMachine in which this Pseudostate is defined. This only applies to Pseudostates of the kind *entryPoint* or *exitPoint*. {Subsets *Element::owner*}

#### Constraints

- [1] An initial vertex can have at most one outgoing transition.  
(self.kind = #initial) **implies**  
    ((self.outgoing->size <= 1))
- [2] History vertices can have at most one outgoing transition.  
((self.kind = #deepHistory) **or** (self.kind = #shallowHistory)) **implies**  
    (self.outgoing->size <= 1)
- [3] In a complete statemachine, a join vertex must have at least two incoming transitions and exactly one outgoing transition.  
(self.kind = #join) **implies**  
    ((self.outgoing->size = 1) and (self.incoming->size >= 2))
- [4] All transitions incoming a join vertex must originate in different regions of an orthogonal state.  
(self.kind = #join)  
    **implies**  
    self.incoming->forAll (t1, t2 | t1<>t2 **implies**  
        (self.stateMachine.LCA(t1.source, t2.source).container.isOrthogonal))
- [5] In a complete statemachine, a fork vertex must have at least two outgoing transitions and exactly one incoming transition.  
(self.kind = #fork) **implies**  
    ((self.incoming->size = 1) **and** (self.outgoing->size >= 2))
- [6] All transitions outgoing a fork vertex must target states in different regions of an orthogonal state.  
(self.kind = #fork)  
    **implies**  
    self.outgoing->forAll (t1, t2 | t1<>t2 **implies**  
        (self.stateMachine.LCA(t1.target, t2.target).  
            container.isOrthogonal))
- [7] In a complete statemachine, a junction vertex must have at least one incoming and one outgoing transition.

(self.kind = #junction) **implies**

((self.incoming->size >= 1) **and** (self.outgoing->size >= 1))

[8] In a complete statemachine, a choice vertex must have at least one incoming and one outgoing transition.

(self.kind = #choice) **implies**

((self.incoming->size >= 1) **and** (self.outgoing->size >= 1))

[9] Pseudostates of kind entryPoint can only be defined in the topmost regions of a StateMachine.

(kind = #entryPoint) **implies** (owner.ocllsKindOf(Region) **and** owner.owner.ocllsKindOf(StateMachine))

[10] Pseudostates of kind exitPoint can only be defined in the topmost regions of a StateMachine.

(kind = #exitPoint) **implies** (owner.ocllsKindOf(Region) **and** owner.owner.ocllsKindOf(StateMachine))

## Semantics

The specific semantics of a Pseudostate depends on the setting of its kind attribute.

- An *initial* pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a region. The initial transition may have an action.
- *deepHistory* represents the most recent active configuration of the composite state that directly contains this pseudostate (e.g., the state configuration that was active when the composite state was last exited). A composite state can have at most one deep history vertex. At most one transition may originate from the history connector to the *default* deep history state. This transition is taken in case the composite state had never been active before. Entry actions of states entered on the path to the state represented by a deep history are performed.
- *shallowHistory* represents the most recent active substate of its containing state (but *not* the substates of that substate). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. At most one transition may originate from the history connector to the *default* shallow history state. This transition is taken in case the composite state had never been active before. Entry actions of states entered on the path to the state represented by a shallow history are performed.
- *join* vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers.
- *fork* vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers.
- *junction* vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as a *merge*). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a *static conditional branch*. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. 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.) Static conditional branches are distinct from dynamic conditional branches that are realized by choice vertices (described below).
- *choice* vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a *dynamic conditional branch*. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed. (To avoid this, it is recommended to define one outgoing transition with the predefined “else” guard for every choice vertex.) Choice vertices should be distinguished from static



branch points that are based on junction points (described above).

- An *entry point* pseudostate is an entry point of a state machine or composite state. In each region of the state machine or composite state it has a single transition to a vertex within the same region.
- An *exit point* pseudostate is an exit point of a state machine or composite state. Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state.
- Entering a *terminate* pseudostate implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate. Entering a terminate pseudostate is equivalent to invoking a DestroyObjectAction.

## Notation

An initial pseudostate is shown as a small solid filled circle (see Figure 15.16). In a region of a classifierBehavior state machine, the transition from an initial pseudostate may be labeled with the trigger event that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any transition from the enclosing state.



**Figure 15.16 - Initial Pseudo State**

A shallowHistory is indicated by a small circle containing an ‘H’ (see Figure 15.17). It applies to the state region that directly encloses it.



**Figure 15.17 - Shallow History**

A deepHistory is indicated by a small circle containing an ‘H\*’ (see Figure 15.18). It applies to the state region that directly encloses it.



**Figure 15.18 - Deep History**

An entry point is shown as a small circle on the border of the state machine diagram or composite state, with the name associated with it (see Figure 15.19).

again ○

**Figure 15.19 - Entry point**

Optionally it may be placed both within the state machine diagram and outside the border of the state machine diagram or composite state.

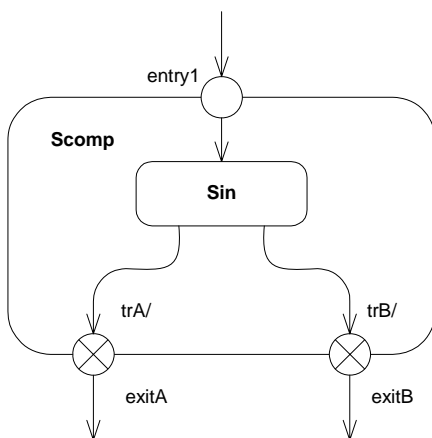
An exit point is shown as a small circle with a cross on the border of the state machine diagram or composite state, with the name associated with it (see Figure 15.20).

⊗ aborted

**Figure 15.20 - Exit point**

Optionally it may be placed both within the state machine diagram or composite state and outside the border of the state machine diagram or composite state.

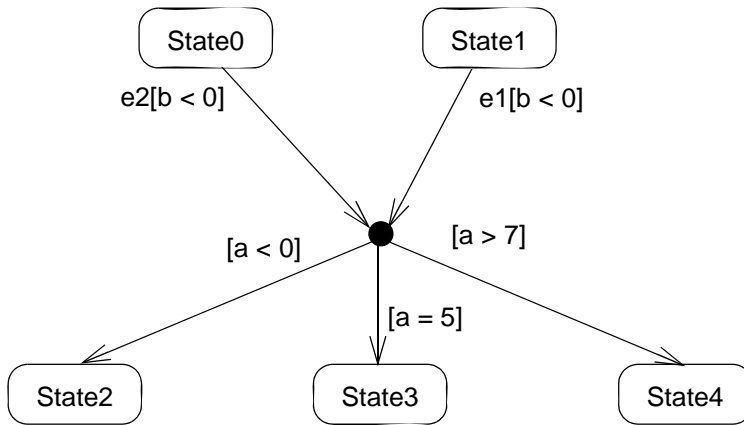
Figure 15.21 illustrates the notation for depicting entry and exit points to composite states (the case of submachine states is illustrated in the corresponding Notation subsection of “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 531).



**Figure 15.21 - Entry and exit points on composite states**

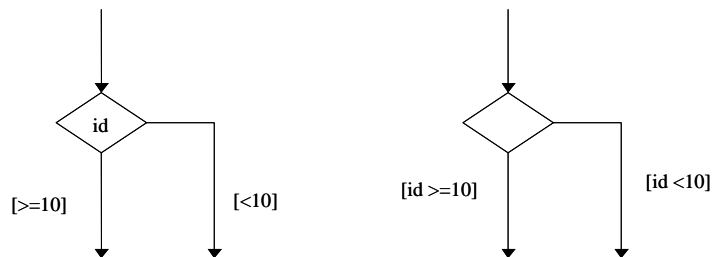
Alternatively, the “bracket” notation shown in Figure 15.9 and Figure 15.10 on page 513 can also be used for the transition-oriented notation.

A junction is represented by a small black circle (see Figure 15.22).



**Figure 15.22 - Junction**

A choice pseudostate is shown as a diamond-shaped symbol as exemplified by Figure 15.23.



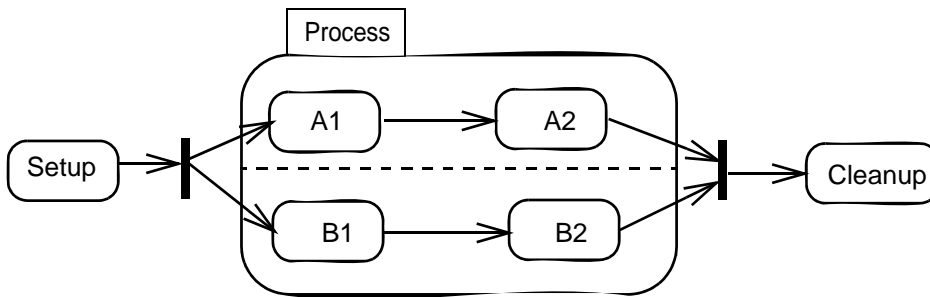
**Figure 15.23 - Choice Pseudo State**

A terminate pseudostate is shown as a cross, see Figure 15.24.



**Figure 15.24 - Terminate node**

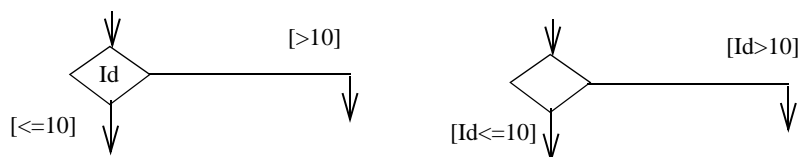
The notation for a fork and join is a short heavy bar (Figure 15.25). The bar may have one or more arrows from source states to the bar (when representing a joint). The bar may have one or more arrows from the bar to states (when representing a fork). A transition string may be shown near the bar.



**Figure 15.25 - Fork and Join**

### Presentation Options

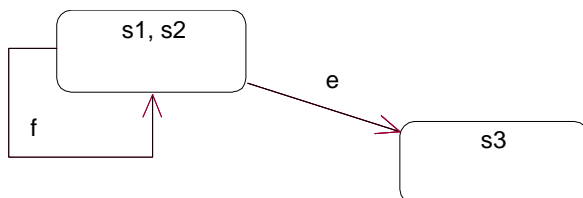
If all guards associated with triggers of transitions leaving a choice Pseudostate are binary expressions that share a common left operand, then the notation for choice Pseudostate may be simplified. The left operand may be placed inside the diamond-shaped symbol and the rest of the Guard expressions placed on the outgoing transitions. This is exemplified in Figure 15.26.



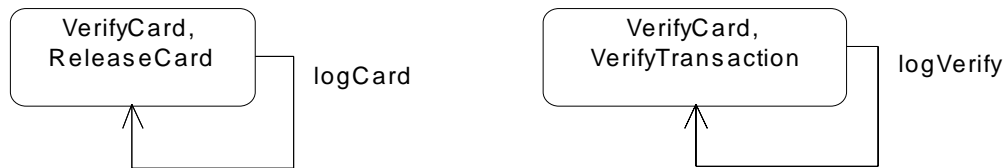
**Figure 15.26 - Alternative Notation for Choice Pseudostate**

Multiple trigger-free and effect-free transitions originating on a set of states and targeting a junction vertex with a single outgoing transition may be presented as a state symbol with a list of the state names and an outgoing transition symbol corresponding to the outgoing transition from the junction.

The special case of the transition from the junction having a history as target may optionally be presented as the target being the state list state symbol. See Figure 15.27 and Figure 15.28 for examples.



**Figure 15.27 - State List Option**



**Figure 15.28 - State Lists**

### Changes from previous UML

- Entry and exit point and terminate Pseudostates have been introduced.
- The semantics of deepHistory has been aligned with shallowHistory in that the containing state does not have to be exited in order for deepHistory to be defined. The implication of this is that deepHistory (as is the case for shallowHistory) can be the target of transitions also within the containing state and not only from states outside.
- The state list presentation option is an extension to UML1.x.

### 15.3.9 PseudostateKind (from BehaviorStateMachines)

PseudostateKind is an enumeration type.

#### Generalizations

None

#### Description

PseudostateKind is an enumeration of the following literal values:

- *initial*
- *deepHistory*
- *shallowHistory*
- *join*
- *fork*
- junction
- *choice*
- entryPoint
- exitPoint
- terminate

#### Attributes

No additional attributes

#### Associations

No additional associations

## Changes from previous UML

EntryPoint, exitPoint, and terminate have been added.

### 15.3.10 Region (from BehaviorStateMachines)

#### Generalizations

- “Namespace (from Kernel)” on page 95
- “RedefinableElement (from Kernel)” on page 125

#### Description

A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

#### Attributes

No additional attributes

#### Associations

- statemachine: StateMachine[0..1] { subsets *NamedElement::namespace* }  
The StateMachine that owns the Region. If a Region is owned by a StateMachine, then it cannot also be owned by a State.
- state: State[0..1] { subsets *NamedElement::namespace* }  
The State that owns the Region. If a Region is owned by a State, then it cannot also be owned by a StateMachine.
- transition: Transition[\*]  
The set of transitions owned by the region. Note that internal transitions are owned by a region, but applies to the source state.
- subvertex: Vertex[\*]  
The set of vertices that are owned by this region.
- extendedRegion: Region[0..1]  
The region of which this region is an extension.
- /redefinitionContext: Classifier[1]  
References the classifier in which context this element may be redefined.

#### Constraints

- [1] A region can have at most one initial vertex.  
self.subvertex->select (v | v.oclIsKindOf(Pseudostate))->  
select(p : Pseudostate | p.kind = #initial)->size() <= 1
- [2] A region can have at most one deep history vertex.  
self.subvertex->select (v | v.oclIsKindOf(Pseudostate))->  
select(p : Pseudostate | p.kind = #deepHistory)->size() <= 1
- [3] A region can have at most one shallow history vertex.  
self.subvertex->select(v | v.oclIsKindOf(Pseudostate))->  
select(p : Pseudostate | p.kind = #shallowHistory)->size() <= 1
- [4] If a Region is owned by a StateMachine, then it cannot also be owned by a State and vice versa.  
(stateMachine->notEmpty() **implies** state->isEmpty()) **and** (state->notEmpty() **implies** stateMachine->isEmpty())

[5] The redefinition context of a region is the nearest containing statemachine.

```
redefinitionContext =  
    let sm = containingStateMachine() in  
    if sm.context->isEmpty() or sm.general->notEmpty() then  
        sm  
    else  
        sm.context  
    endif
```

### Additional constraints

- [1] The query `isRedefinitionContextValid()` specifies whether the redefinition contexts of a region are properly related to the redefinition contexts of the specified region to allow this element to redefine the other. The containing statemachine/state of a redefining region must redefine the containing statemachine/state of the redefined region.
- [2] The query `isConsistentWith()` specifies that a redefining region is consistent with a redefined region provided that the redefining region is an extension of the redefined region (i.e., it adds vertices and transitions and it redefines states and transitions of the redefined region).

### Additional operations

- [1] The operation `containingStateMachine()` returns the state machine in which this Region is defined.

```
context Region::containingStateMachine() : StateMachine  
post: result = if stateMachine->isEmpty() then  
    state.containingStateMachine()  
else  
    stateMachine
```

### Semantics

The semantics of regions is tightly coupled with states or state machines having regions, and it is therefore defined as part of the semantics for state and state machine.

When a composite state or state machine is extended, each inherited region may be extended, and regions may be added.

### Notation

A composite state or state machine with regions is shown by tiling the graph region of the state/state machine using dashed lines to divide it into regions. Each region may have an optional name and contains the nested disjoint states and the transitions between these. The text compartments of the entire state are separated from the orthogonal regions by a solid line.

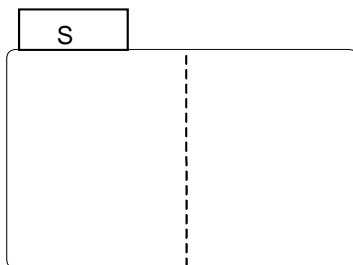


Figure 15.29 - Notation for composite state/state machine with regions

A composite state or state machine with just one region is shown by showing a nested state diagram within the graph region.

In order to indicate that an inherited region is extended, the keyword «extended» is associated with the name of the region.

### 15.3.11 State (from BehaviorStateMachines, ProtocolStateMachines)

A state models a situation during which some (usually implicit) invariant condition holds.

#### Generalizations

- “Namespace (from Kernel)” on page 95
- “RedefinableElement (from Kernel)” on page 125
- “Vertex (from BehaviorStateMachines)” on page 562

#### Description

##### *State in Behavioral State machines*

A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behavior (i.e., the model element under consideration enters the state when the behavior commences and leaves it as soon as the behavior is completed).

The following kinds of states are distinguished:

- Simple state,
- composite state, and
- submachine state.

A composite state is either a simple composite state (with just one region) or an orthogonal state (with more than one region).

##### *Simple state*

A simple state is a state that does not have substates (i.e., it has no regions and it has no submachine state machine).

##### *Composite state*

A composite state either contains one region or is decomposed into two or more orthogonal *regions*. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions. A given state may only be decomposed in one of these two ways.

Any state enclosed within a region of a composite state is called a *substate* of that composite state. It is called a *direct substate* when it is not contained by any other state; otherwise, it is referred to as an *indirect substate*.

Each region of a composite state may have an initial pseudostate and a final state. A transition to the enclosing state represents a transition to the initial pseudostate in each region. A newly-created object takes its topmost default transitions, originating from the topmost initial pseudostates of each region.



A transition to a final state represents the completion of behavior in the enclosing region. Completion of behavior in all orthogonal regions represents completion of behavior by the enclosing state and triggers a completion event on the enclosing state. Completion of the topmost regions of an object corresponds to its termination.

An entry pseudostate is used to join an external transition terminating on that entry point to an internal transition emanating from that entry point. An exit pseudostate is used to join an internal transition terminating on that exit point to an external transition emanating from that exit point. The main purpose of such entry and exit points is to execute the state entry and exit actions respectively in between the actions that are associated with the joined transitions.

#### *Semantic variation point (default entry rule)*

If a transition terminates on an enclosing state and the enclosed regions do not have an initial pseudostate, the interpretation of this situation is a semantic variation point. In some interpretations, this is considered an ill-formed model. That is, in those cases the initial pseudostate is mandatory.

An alternative interpretation allows this situation and it means that, when such a transition is taken, the state machine stays in the composite state, *without entering any of the regions or their substates*.

#### *Submachine state*

A submachine state specifies the insertion of the specification of a submachine state machine. The state machine that contains the submachine state is called the *containing* state machine. The same state machine may be a submachine more than once in the context of a single containing state machine.

A submachine state is semantically equivalent to a composite state. The regions of the submachine state machine are the regions of the composite state. The entry, exit, and behavior actions and internal transitions are defined as part of the state. Submachine state is a decomposition mechanism that allows factoring of common behaviors and their reuse.

Transitions in the containing state machine can have entry/exit points of the inserted state machine as targets/sources.

#### *State in Protocol State machines*

The states of protocol state machines are exposed to the users of their context classifiers. A protocol state represents an exposed stable situation of its context classifier: When an instance of the classifier is not processing any operation, users of this instance can always know its state configuration.

#### **Attributes**

- `/isComposite` : Boolean [1]     A state with `isComposite=true` is said to be a *composite state*. A composite state is a state that contains at least one region.
- `/isOrthogonal`: Boolean [1]     A state with `isOrthogonal=true` is said to be an *orthogonal composite state*. An orthogonal composite state contains two or more regions.
- `/isSimple`: Boolean [1]     A state with `isSimple=true` is said to be a *simple state*. A simple state does not have any regions and it does not refer to any submachine state machine.
- `/isSubmachineState`: Boolean [1]     A state with `isSubmachineState=true` is said to be a *submachine state*. Such a state refers to a state machine (submachine).

## Associations

### *Package BehaviorStateMachines*

- **connection:** ConnectionPointReference [\*]  
The entry and exit connection points used in conjunction with this (submachine) state, i.e., as targets and sources, respectively, in the region with the submachine state. A connection point reference references the corresponding definition of a connection point pseudostate in the statemachine referenced by the submachinestate.
- **connectionPoint:** Pseudostate [\*]  
The entry and exit pseudostates of a composite state. These can only be entry or exit Pseudostates, and they must have different names. They can only be defined for composite states.
- **deferrableTrigger:** Trigger [\*]  
A list of triggers that are candidates to be retained by the state machine if they trigger no transitions out of the state (not consumed). A deferred trigger is retained until the state machine reaches a state configuration where it is no longer deferred.
- **doActivity:** Behavior[0..1]  
An optional behavior that is executed while being in the state. The execution starts when this state is entered, and stops either by itself or when the state is exited whichever comes first.
- **entry:** Behavior[0..1]  
An optional behavior that is executed whenever this state is entered regardless of the transition taken to reach the state. If defined, entry actions are always executed to completion prior to any internal behavior or transitions performed within the state.
- **exit:** Behavior[0..1]  
An optional behavior that is executed whenever this state is exited regardless of which transition was taken out of the state. If defined, exit actions are always executed to completion only after all internal activities and transition actions have completed execution.
- **redefinedState:** State[0..1]  
The state of which this state is a redefinition.
- **region:** Region[\*] {subsets *ownedMember*}  
The regions owned directly by the state.
- **submachine:** StateMachine[0..1]  
The state machine that is to be inserted in place of the (submachine) state.
- **stateInvariant:** Constraint [0..1]  
Specifies conditions that are always true when this state is the current state. In protocol state machines, state invariants are additional conditions to the preconditions of the outgoing transitions, and to the postcondition of the incoming transitions.
- **/redefinitionContext:** Classifier[1]  
References the classifier in which context this element may be redefined.

## Constraints

- [1] There have to be at least two regions in an orthogonal composite state.  
(self.isOrthogonal) **implies**  
(self.region->size >= 2)
- [2] Only submachine states can have connection point references.  
isSubmachineState **implies** connection->notEmpty ( )

- [3] The connection point references used as destinations/sources of transitions associated with a submachine state must be defined as entry/exit points in the submachine state machine.
- [4] A state is not allowed to have both a submachine and regions.  
**isComposite** implies not **isSubmachineState**
- [5] A simple state is a state without any regions.  
`isSimple = content.isEmpty()`
- [6] A composite state is a state with at least one region.  
`isComposite = content.notEmpty()`
- [7] An orthogonal state is a composite state with at least 2 regions.  
`isOrthogonal = (region->size () > 1)`
- [8] Only submachine states can have a reference statemachine.  
`isSubmachineState = submachine.notEmpty()`
- [9] The redefinition context of a state is the nearest containing statemachine.  
`redefinitionContext =`  
`let sm = containingStateMachine() in`  
`if sm.context->isEmpty() or sm.general->notEmpty() then`  
`sm`  
`else`  
`sm.context`  
`endif`
- [10] Only composite states can have entry or exit pseudostates defined.  
`connectionPoint->notEmpty() implies isComposite`
- [11] Only entry or exit pseudostates can serve as connection points.  
`connectionPoint->forall(cp|cp.kind = #entry or cp.kind = #exit)`

## Additional Operations

- [1] The query `isRedefinitionContextValid()` specifies whether the redefinition contexts of a state are properly related to the redefinition contexts of the specified state to allow this element to redefine the other. The containing region of a redefining state must redefine the containing region of the redefined state.
- [2] The query `isConsistentWith()` specifies that a redefining state is consistent with a redefined state provided that the redefining state is an extension of the redefined state: A simple state can be redefined (extended) to become a composite state (by adding a region) and a composite state can be redefined (extended) by adding regions and by adding vertices, states, and transitions to inherited regions. All states may add or replace entry, exit, and “doActivity” actions.
- [3] The query `containingStateMachine()` returns the state machine that contains the state either directly or transitively.  
**context** `State::containingStateMachine() : StateMachine`  
**post:** `result = container.containingStateMachine()`

## Semantics

### *States in general*

The following applies to states in general. Special semantics applies to composite states and submachine states.

### *Active states*

A state can be active or inactive during execution. A state becomes *active* when it is entered as a result of some transition, and becomes *inactive* if it is exited as a result of a transition. A state can be exited and entered as a result of the same transition (e.g., self transition).

### *State entry and exit*

Whenever a state is entered, it executes its entry behavior *before* any other action is executed. Conversely, whenever a state is exited, it executes its exit behavior as the final step prior to leaving the state.

### *Behavior in state (do-activity)*

The behavior represents the execution of a behavior, that occurs while the state machine is in the corresponding state. The behavior starts executing upon entering the state, following the entry behavior. If the behavior completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition (see below) the state will be exited. Upon exit, the behavior is terminated before the exit behavior is executed. If the state is exited as a result of the firing of an outgoing transition before the completion of the behavior, the behavior is aborted prior to its completion.

### *Deferred events*

A state may specify a set of event types that may be *deferred* in that state. An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event pool while another non-deferred event is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.

### *State redefinition*

A state may be redefined. A simple state can be redefined (extended) to become a composite state (by adding a region) and a composite state can be redefined (extended) by adding regions and by adding vertices, states, entry/exit/do activities (if the general state has none), and transitions to inherited regions. The redefinition of a state applies to the whole state machine. For example, if a state list as part of the extended state machine includes a state that is redefined, then the state list for the extension state machine includes the redefined state.

## *Composite state*

### *Active state configurations*

In a hierarchical state machine more than one state can be active at the same time. If the state machine is in a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active. Furthermore, since the state machine as a whole and some of the composite states in this hierarchy may be orthogonal (i.e., containing regions), the current active “state” is actually represented by a set of trees of states starting with the top-most states of the root regions down to the innermost active substate. We refer to such a state tree as a *state configuration*.

Except during transition execution, the following invariants always apply to state configurations:

- If a composite state is active and not orthogonal, at most one of its substates is active.
- If the composite state is active and orthogonal, all of its regions are active, with at most one substate in each region.

### *Entering a non-orthogonal composite state*

Upon entering a composite state, the following cases are differentiated:

- *Default entry*: Graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state. In this case, the default entry rule is applied (see *Semantic variation point (default entry rule)*). If there is a guard on the trigger of the transition, it must be enabled (true). (A disabled initial transition is an ill-defined execution state and its handling is not defined.) The entry behavior of the composite state is executed before the behavior associated with the initial transition.
- *Explicit entry*: If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate.
- *Shallow history entry*: If the transition terminates on a shallow history pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the *default history state* is entered. This is the substate that is target of the transition originating from the history pseudostate. (If no such transition is specified, the situation is ill-defined and its handling is not defined.) If the active substate determined by history is a composite state, then it proceeds with its default entry.
- *Deep history entry*: The rule here is the same as for shallow history except that the rule is applied recursively to all levels in the active state configuration below this one.
- *Entry point entry*: If a transition enters a composite state through an entry point pseudostate, then the entry behavior is executed before the action associated with the internal transition emanating from the entry point.

#### *Entering an orthogonal composite state*

Whenever an orthogonal composite state is entered, each one of its orthogonal regions is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of a fork), these regions are entered explicitly and the others by default.

#### *Exiting non-orthogonal state*

When exiting from a composite state, the active substate is exited recursively. This means that the exit activities are executed in sequence starting with the innermost active state in the current state configuration.

If, in a composite state, the exit occurs through an exit point pseudostate the exit behavior of the state is executed *after* the behavior associated with the transition incoming to the exit point.

#### *Exiting an orthogonal state*

When exiting from an orthogonal state, each of its regions is exited. After that, the exit activities of the state are executed.

#### *Deferred events*

Composite states introduce potential event deferral conflicts. Each of the substates may defer or consume an event, potentially conflicting with the composite state (e.g., a substate defers an event while the composite state consumes it, or vice versa). In case of a composite orthogonal state, substates of orthogonal regions may also introduce deferral conflicts. The conflict resolution follows the triggering priorities, where nested states override enclosing states. In case of a conflict between states in different orthogonal regions, a consumer state overrides a deferring state.

#### *Submachine state*

A submachine state is semantically equivalent to the composite state defined by the referenced state machine. Entering and leaving this composite state is, in contrast to an ordinary composite state, via entry and exit points.

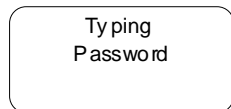
A submachine composite state machine can be entered via its default (initial) pseudostate or via any of its entry points (i.e., it may imply entering a non-orthogonal or an orthogonal composite state with regions). Entering via the initial pseudostate has the same meaning as for ordinary composite states. An entry point is equivalent with a junction pseudostate (fork in case the composite state is orthogonal): Entering via an entry point implies that the entry behavior of the composite state is executed, followed by the (partial) transition(s) from the entry point to the target state(s) within the composite state. As for default initial transitions, guards associated with the triggers of these entry point transitions must evaluate to true in order for the specification not to be ill-formed.

Similarly, it can be exited as a result of reaching its final state, by a “group” transition that applies to all substates in the submachine state composite state, or via any of its exit points. Exiting via a final state or by a group transition has the same meaning as for ordinary composite states. An exit point is equivalent with a junction pseudostate (join in case the composite state is orthogonal): Exiting via an exit point implies that first behavior of the transition with the exit point as target is executed, followed by the exit behavior of the composite state.

## Notation

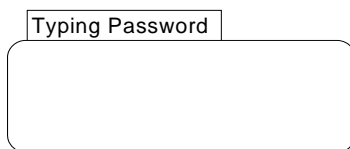
### *States in general*

A state is in general shown as a rectangle with rounded corners, with the state name shown inside the rectangle.



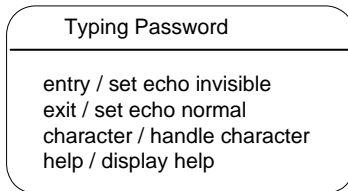
**Figure 15.30 - State**

Optionally, it may have an attached name tab, see Figure 15.31. 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 orthogonal regions, but may be used in other cases as well. The state in Figure 15.25 on page 527 illustrates the use of the name tab.



**Figure 15.31 - State with name tab**

A state may be subdivided into multiple compartments separated from each other by a horizontal line, see Figure 15.32.



**Figure 15.32 - State with compartments**

The compartments of a state are:

- name compartment
- internal activities compartment
- internal transitions compartment

A composite state has in addition a

- decomposition compartment

Each of these compartments is described below.

- 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, unless control icons (page 557) are used to show a transition oriented view of the state machine. Name compartments should not be used if a name tab is used and vice versa.

In case of a submachine state, the name of the referenced state machine is shown as a string following ‘:’ after the name of the state.

- Internal activities compartment

This compartment holds a list of internal actions or state (do) activities (behaviors) that are performed while the element is in the state.

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

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

- *entry* — This label identifies a behavior, specified by the corresponding expression, which is performed upon entry to the state (entry behavior).
- *exit* — This label identifies a behavior, specified by the corresponding expression, that is performed upon exit from the state (exit behavior).

- *do* — This label identifies an ongoing behavior (“do activity”) that is performed as long as the modeled element is in the state or until the computation specified by the expression is completed (the latter may result in a completion event being generated).
- Internal transition compartment

This compartment contains a list of internal transitions, where each item has the form as described for Trigger.

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 expression through the current event variable.

### *Composite state*

- decomposition compartment

This compartment shows its composition structure in terms of regions, states, and transition. In addition to the (optional) name and internal transition compartments, the state may have an additional compartment that contains a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic region.

In some cases, it is convenient to hide the decomposition of a composite state. For example, there may be a large number of states nested inside a composite state and they 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 (see Figure 15.34). 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 state machine diagram. 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.

A composite state may have one or more entry and exit points on its outside border or in close proximity of that border (inside or outside).



## Examples

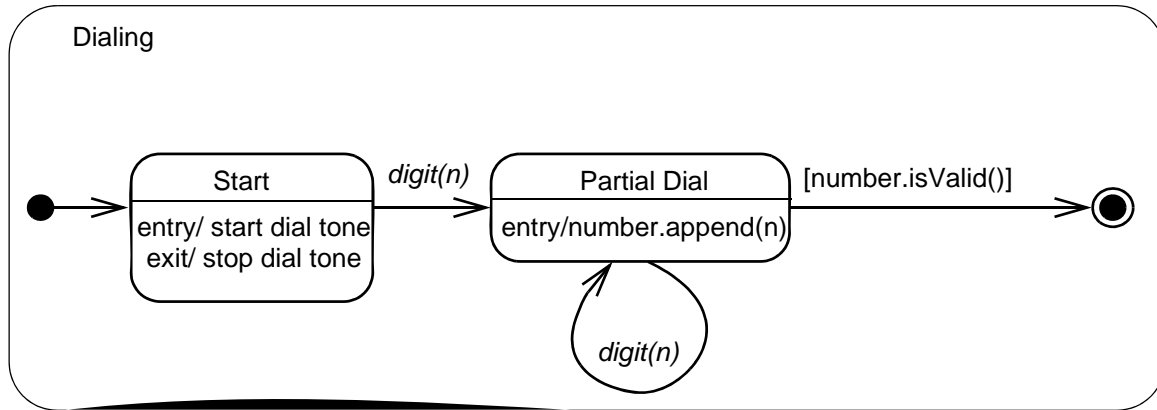


Figure 15.33 - Composite state with two states

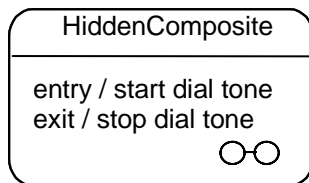
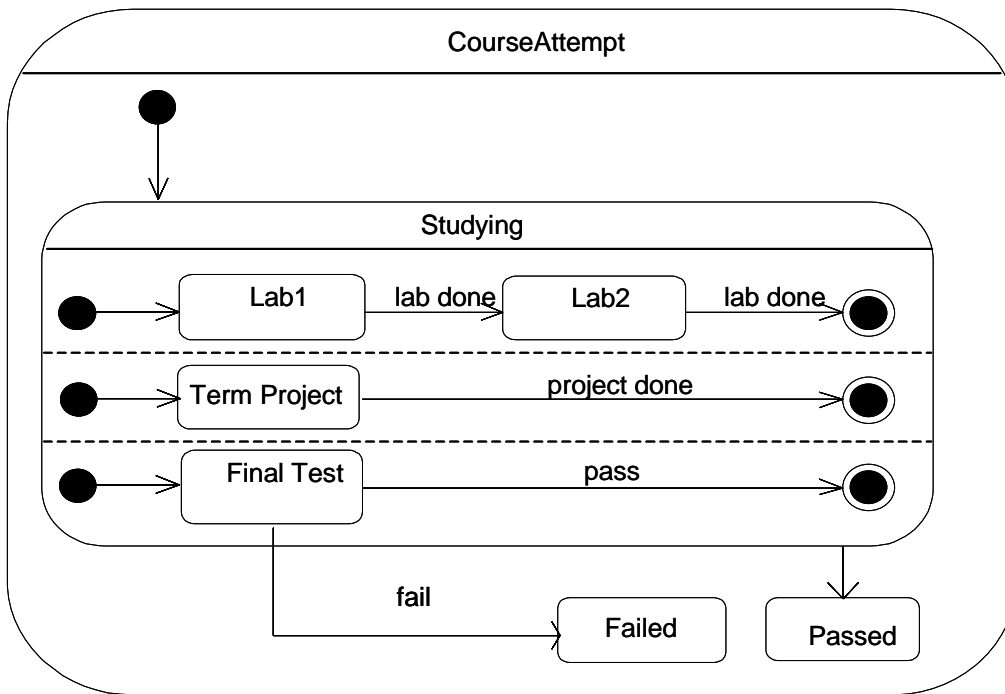


Figure 15.34 - Composite State with hidden decomposition indicator icon



**Figure 15.35 - Orthogonal state with regions**

#### *Submachine state*

The submachine state is depicted as a normal state where the string in the name compartment has the following syntax:

<state name> ‘:’ <name of referenced state machine>

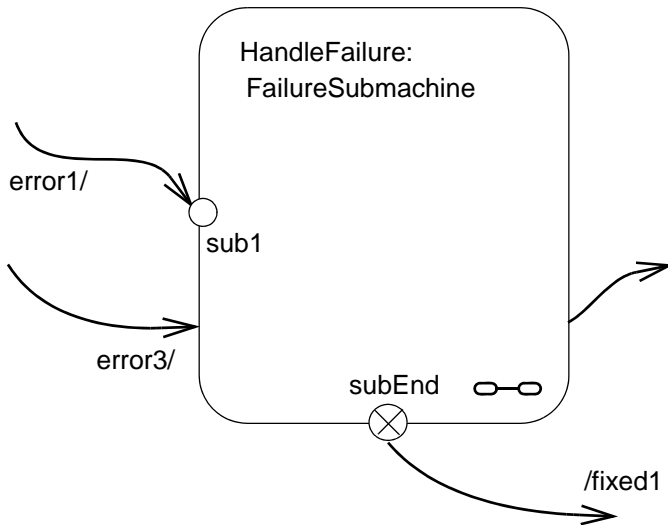
The submachine state symbol may contain the *references* to one or more entry points and to one or more exit points. The notation for these connection point references are entry/exit point pseudostates on the border of the submachine state. The names are the names of the corresponding entry/exit points *defined* within the referenced state machine. See (“ConnectionPointReference (from BehaviorStateMachines)” on page 511).

If the substate machine is entered through its default initial pseudostate or if it is exited as a result of the completion of the submachine, it is not necessary to use the entry/exit point notation. Similarly, an exit point 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 the entry and exit points being part of different transitions.

## Examples

The diagram in Figure 15.36 shows a fragment from a state machine diagram in which a sub state machine (the FailureSubmachine) is referenced. The actual sub state machine is defined in some enclosing or imported name space.



**Figure 15.36 - Submachine State**

In the above example, the transition triggered by event “error1” will terminate on entry point “sub1” of the FailureSubmachine state machine. The “error3” transition implies taking of the default transition of the FailureSubmachine.

The transition emanating from the “subEnd” exit point of the submachine will execute the “fixed1” behavior in addition to what is executed within the HandleFailure state machine. This transition must have been triggered within the HandleFailure state machine. 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.

Note that the same notation would apply to composite states with the exception that there would be no reference to a state machine in the state name.

Figure 15.37 is an example of a state machine defined with two exit points. The entry and exit points may also be shown on the frame or outside the frame (but still associated with it), and not only within the state graph.

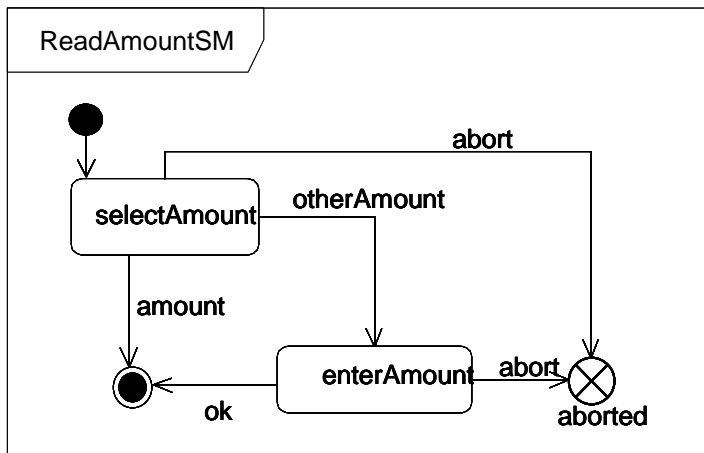


Figure 15.37 - State machine with exit point as part of the state graph

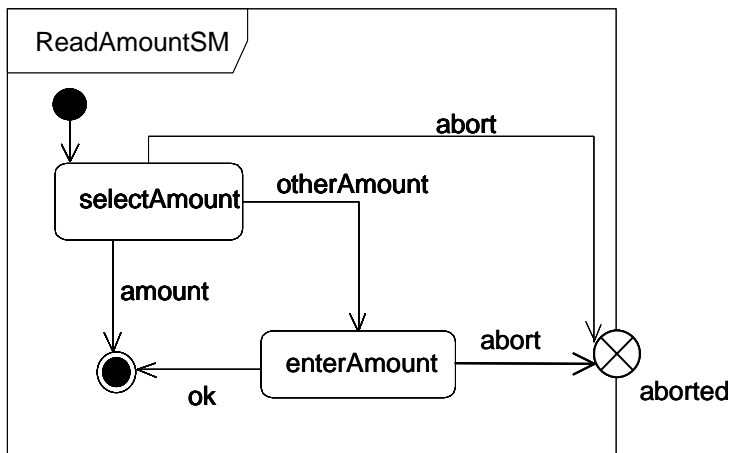
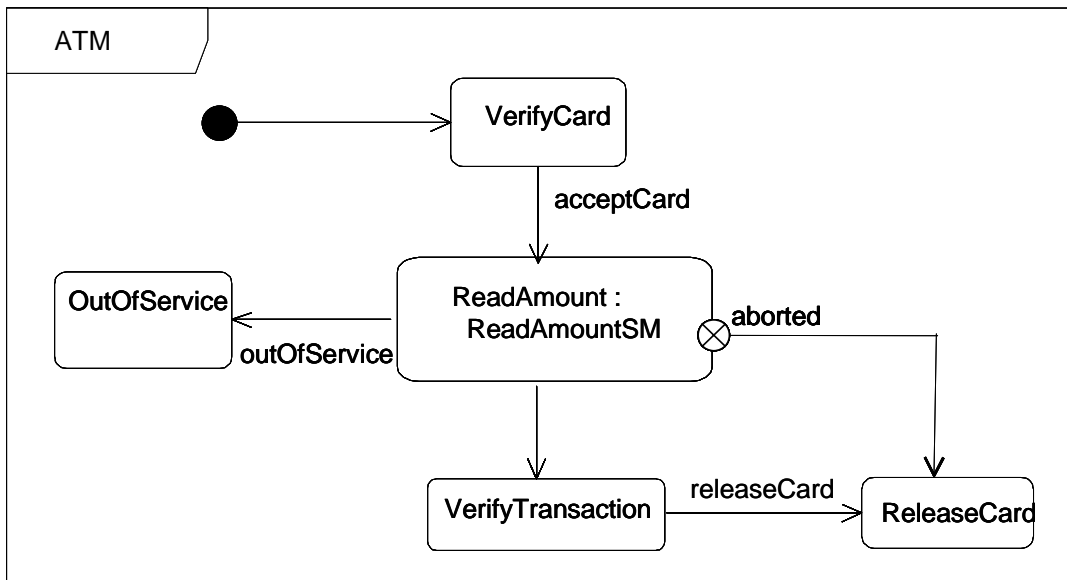


Figure 15.38 - State machine with exit point on the border of the statemachine

In Figure 15.39 this state machine is referenced in a submachine state, and the presentation option with the exit points on the state symbol is shown.

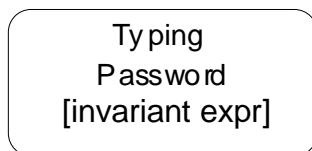


**Figure 15.39 - SubmachineState with usage of exit point**

An example of the notation for entry and exit points for composite states is shown in Figure 15.21 on page 525.

#### *Notation for protocol state machines*

The two differences that exist for state in protocol state machine, versus states in behavioral state machine, are as follows: Several features in behavioral state machine do not exist for protocol state machines (entry, exit, do); States in protocol state machines can have an invariant. The textual expression of the invariant will be represented by placing it after or under the name of the state, surrounded by square brackets.



**Figure 15.40 - State with invariant - notation**

#### **Rationale**

Submachine states with usages of entry and exit points defined in the corresponding state machine have been introduced in order for state machines with submachines to scale and in order to provide encapsulation.

### 15.3.12 StateMachine (from BehaviorStateMachines)

State machines can be used to express the behavior of part of a system. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine.

#### Generalizations

- “Behavior (from BasicBehaviors)” on page 416

#### Description

A state machine owns one or more regions, which in turn own vertices and transitions.

The behavored classifier context owning a state machine defines which signal and call triggers are defined for the state machine, and which attributes and operations are available in activities of the state machine. Signal triggers and call triggers for the state machine are defined according to the receptions and operations of this classifier.

As a kind of behavior, a state machine may have an associated behavioral feature (specification) and be the method of this behavioral feature. In this case the state machine specifies the behavior of this behavioral feature. The parameters of the state machine in this case match the parameters of the behavioral feature and provide the means for accessing (within the state machine) the behavioral feature parameters.

A state machine without a context classifier may use triggers that are independent of receptions or operations of a classifier, i.e., either just signal triggers or call triggers based upon operation template parameters of the (parameterized) statemachine.

#### Attributes

No additional attributes

#### Associations

- |                                                      |                                                                                                                                                |
|------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| • region: Region[1..*] {subsets <i>ownedMember</i> } | The regions owned directly by the state machine.                                                                                               |
| • connectionPoint: Pseudostate[*]                    | The connection points defined for this state machine. They represent the interface of the state machine when used as part of submachine state. |
| • extendedStateMachine: StateMachine[*]              | The state machines of which this is an extension.                                                                                              |

#### Constraints

- [1] The classifier context of a state machine cannot be an interface.  
context->notEmpty() **implies not** context.ocllsKindOf(Interface)
- [2] The context classifier of the method state machine of a behavioral feature must be the classifier that owns the behavioral feature.  
specification->notEmpty() **implies** (context->notEmpty() **and** specification->featuringClassifier->exists (c | c = context))
- [3] The connection points of a state machine are pseudostates of kind entry point or exit point.  
conectionPoint->forAll (c | c.kind = #entryPoint **or** c.kind = #exitPoint)
- [4] A state machine as the method for a behavioral feature cannot have entry/exit connection points.

specification->notEmpty() **implies** connectionPoint->isEmpty()

## Additional Operations

- [1] The operation LCA(s1,s2) returns an orthogonal state or region that is the least common ancestor of states s1 and s2, based on the statemachine containment hierarchy.
- [2] The query ancestor(s1, s2) checks whether s2 is an ancestor state of state s1.  
**context** StateMachine::ancestor (s1 : State, s2 : State) : Boolean  
result = **if** (s2 = s1) **then**  
    true  
    **else if** (s1.container->isEmpty) **then**  
        true  
    **else if** (s2.container->isEmpty) **then**  
        false  
    **else** (ancestor (s1, s2.container))
- [3] The query isRedefinitionContextValid() specifies whether the redefinition contexts of a statemachine are properly related to the redefinition contexts of the specified statemachine to allow this element to redefine the other. The containing classifier of a redefining statemachine must redefine the containing classifier of the redefined statemachine.
- [4] The query isConsistentWith() specifies that a redefining state machine is consistent with a redefined state machine provided that the redefining state machine is an extension of the redefined state machine: Regions are inherited and regions can be added, inherited regions can be redefined. In case of multiple redefining state machines, extension implies that the redefining state machine gets orthogonal regions for each of the redefined state machines.

## Semantics

The event pool for the state machine is the event pool of the instance according to the behavioed context classifier, or the classifier owning the behavioral feature for which the state machine is a method.

### *Event processing - run-to-completion step*

Event occurrences are detected, dispatched, and then processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.

The semantics of event occurrence processing is based on the *run-to-completion* assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.

Run-to-completion may be implemented in various ways. For active classes, it may be realized by an event-loop running in its own thread, and that reads event occurrences from a pool. For passive classes it may be implemented as a monitor.

The processing of a single event occurrence by a state machine is known as a *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion step* is the passage between two state configurations of the state machine.

The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step.

When an event occurrence is detected and dispatched, it may result in one or more transitions being enabled for firing. If no transition is enabled and the event (type) is not in the deferred event list of the current state configuration, the event occurrence is discarded and the run-to-completion step is completed.

In the presence of orthogonal regions it is possible to fire multiple transitions as a result of the same event occurrence — as many as one transition in each region in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. Which of the enabled transitions actually fire is determined by the transition selection algorithm described below. The order in which selected transitions fire is not defined.

Each orthogonal region in the active state configuration that is not decomposed into orthogonal regions (i.e., “bottom-level” region) can fire at most one transition as a result of the current event occurrence. When all orthogonal regions have finished executing the transition, the current event occurrence is fully consumed, and the run-to-completion step is completed.

During a transition, a number of actions may be executed. If such an action is a synchronous operation invocation on an object executing a state machine, then the transition step is not completed until the invoked object complete its run-to-completion step.

### *Run-to-completion and concurrency*

It is possible to define state machine semantics by allowing the run-to-completion steps to be applied orthogonally to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the steps taken by orthogonal regions in the active state configuration.

In case of active objects, where each object has its own thread of execution, it is very important to clearly distinguish the notion of run to completion from the concept of thread pre-emption. Namely, run-to-completion event handling is performed by a thread that, in principle, *can* be pre-empted and its execution suspended in favor of another thread executing on the same processing node. (This is determined by the scheduling policy of the underlying thread environment — no assumptions are made about this policy.) When the suspended thread is assigned processor time again, it resumes its event processing from the point of pre-emption and, eventually, completes its event processing.

### *Conflicting transitions*

It was already noted that it is possible for more than one transition to be enabled within a state machine. If that happens, then such transitions may be in *conflict* with each other. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step.

Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed.

An internal transition in a state conflicts only with transitions that cause an exit from that state.

### *Firing priorities*

In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an *implicit* priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.

The priority of a transition is defined based on its source state. The priority of joined transitions is based on the priority of the transition with the most transitively nested source state.



In general, if  $t_1$  is a transition whose source state is  $s_1$ , and  $t_2$  has source  $s_2$ , then:

- If  $s_1$  is a direct or transitively nested substate of  $s_2$ , then  $t_1$  has higher priority than  $t_2$ .
- If  $s_1$  and  $s_2$  are not in the same state configuration, then there is no priority difference between  $t_1$  and  $t_2$ .

#### *Transition selection algorithm*

The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions:

- All transitions in the set are enabled.
- There are no conflicting transitions within the set.
- There is no transition outside the set that has higher priority than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out).

This can be easily implemented by a greedy selection algorithm, with a straightforward traversal of the active state configuration. States in the active state configuration are traversed starting with the innermost nested simple states and working outwards. For each state at a given level, all originating transitions are evaluated to determine if they are enabled. This traversal guarantees that the priority principle is not violated. The only non-trivial issue is resolving transition conflicts across orthogonal states on all levels. This is resolved by terminating the search in each orthogonal state once a transition inside any one of its components is fired.

#### *StateMachine extension*

A state machine is generalizable. A specialized state machine is an extension of the general state machine, in that regions, vertices, and transitions may be added; regions and states may be redefined (extended: simple states to composite states and composite states by adding states and transitions); and transitions can be redefined.

As part of a classifier generalization, the classifierBehavior state machine of the general classifier and the method state machines of behavioral features of the general classifier can be redefined (by other state machines). These state machines may be specializations (extensions) of the corresponding state machines of the general classifier or of its behavioral features.

A specialized state machine will have all the elements of the general state machine, and it may have additional elements. Regions may be added. Inherited regions may be redefined by extension: States and vertices are inherited, and states and transitions of the regions of the state machine may be redefined.

A simple state can be redefined (extended) to a composite state, by adding one or more regions.

A composite state can be redefined (extended) by either extending inherited regions or by adding regions as well as by adding entry and exit points. A region is extended by adding vertices, states, and transitions and by redefining states and transitions.

A submachine state may be redefined. The submachine state machine may be replaced by another submachine state machine, provided that it has the same entry/exit points as the redefined submachine state machine, but it may add entry/exit points.

Transitions can have their content and target state replaced, while the source state and trigger is preserved.

In case of multiple general classifiers, extension implies that the extension state machine gets orthogonal regions for each of the state machines of the general classifiers in addition to the one of the specific classifier.

## Notation

A state machine 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 connect them or by control icons representing the actions of the behavior on the transition (page 557).

The association between a state machine and its context classifier or behavioral feature does not have a special notation.

A state machine that is an extension of the state machine in a general classifier will have the keyword «extended» associated with the name of the state machine.

The default notation for classifier is used for denoting state machines. The keyword is «statemachine».

Inherited states are drawn with dashed lines or gray-toned lines.

## Presentation option

Inherited states are drawn with gray-toned lines.

## Examples

Figure 15.41 is an example statemachine diagram for the state machine for simple telephone object. In addition to the initial state, the state machine has an entry point called activeEntry, and in addition to the final state, it has an exit point called aborted.

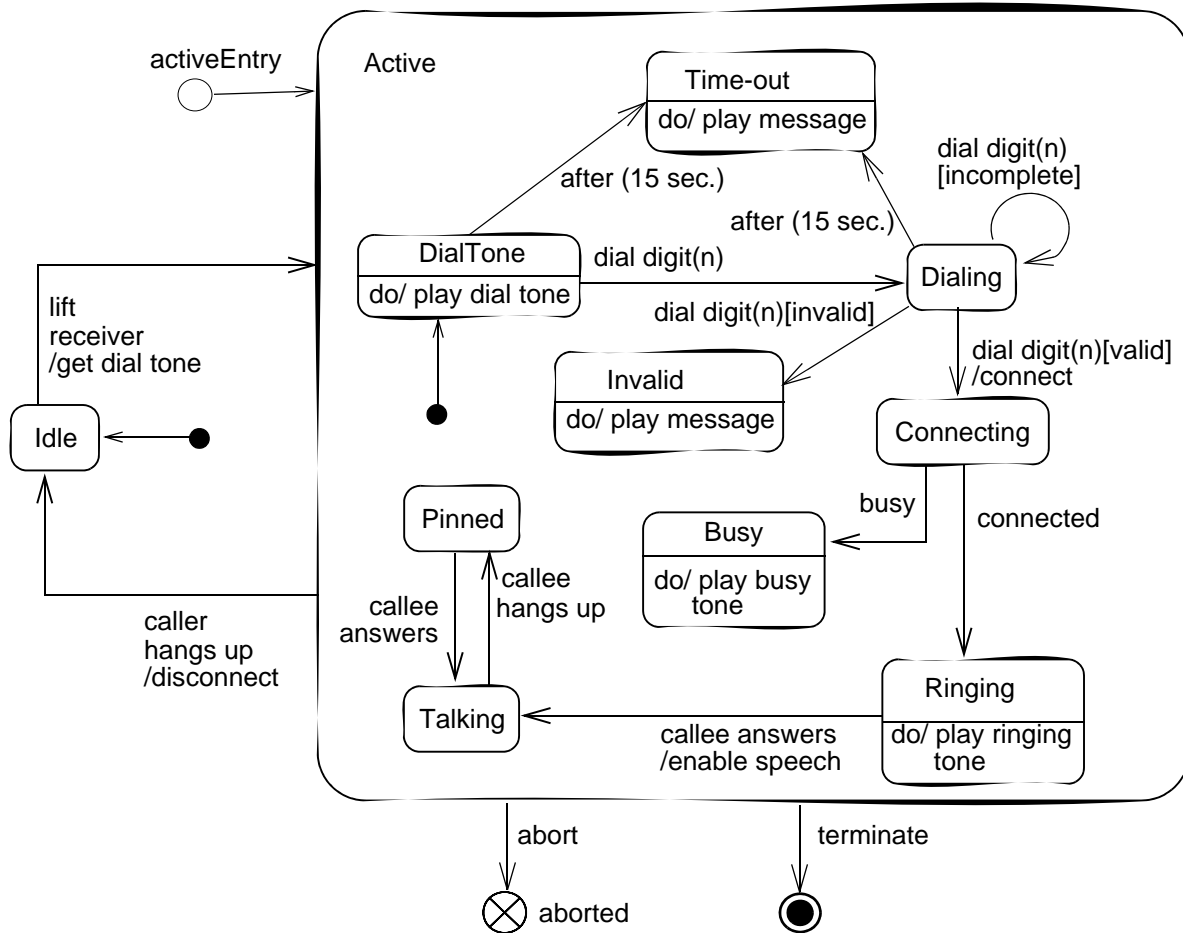
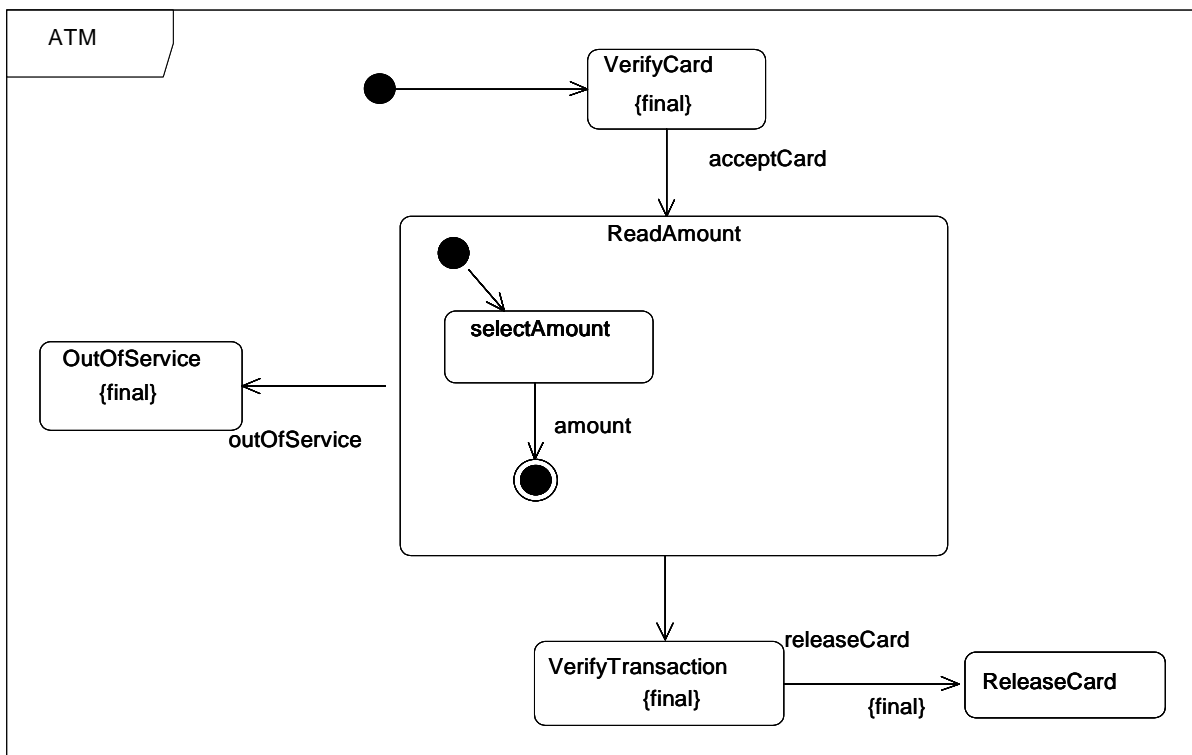


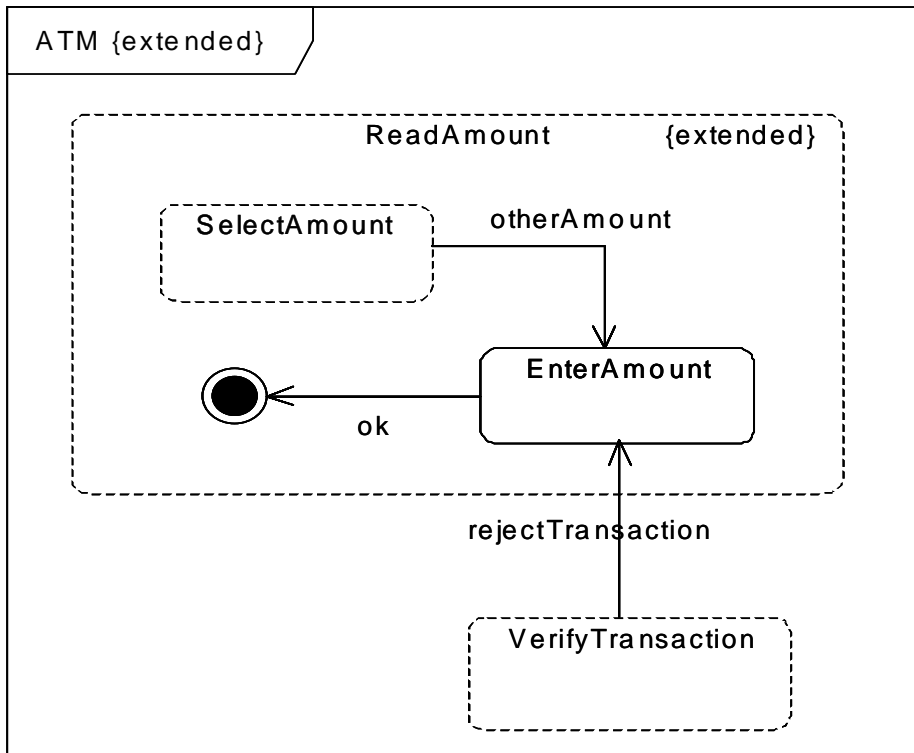
Figure 15.41 - State machine diagram representing a state machine

As an example of state machine specialization, the states `VerifyCard`, `OutOfService`, and `VerifyTransaction` in the ATM state machine in Figure 15.42 have been specified as `{final}`, which means that they cannot be redefined (i.e., extended) in specializations of `ATM`. The other states can be redefined. The `(verifyTransaction,releaseCard)` transition has also been specified as `{final}`, meaning that the effect behavior and the target state cannot be redefined.



**Figure 15.42 - A general state machine**

In Figure 15.43 a specialized ATM (which is the statemachine of a class that is a specialization of the class with the ATM statemachine of Figure 15.42) is defined by extending the composite state by adding a state and a transition, so that users can enter the desired amount. In addition a transition is added from an inherited state to the newly introduced state.



**Figure 15.43 - An extended state machine**

### Rationale

The rationale for statemachine extension is that it shall be possible to define the redefined behavior of a special classifier as an extension of the behavior of the general classifier.

### Changes from previous UML

State machine extension is an extension of 1.x. In 1.x, state machine generalization is only explained informally through a note.

### Rationale

State machines are used for the definition of behavior (for example, classes that are generalizable). As part of the specialization of a class it is desirable also to specialize the behavior definition.

### 15.3.13 TimeEvent (from BehaviorStateMachines)

#### Generalizations

- "TimeEvent (from Communications, SimpleTime)" on page 438 (*merge increment*)

## Description

Extends TimeEvent to be defined relative to entering the current state of the executing state machine.

## Constraints

[1] The starting time for a relative time event may only be omitted for a time event that is the trigger of a state machine.

## Semantics

If the deadline expression is relative and no explicit starting time is defined, then it is relative to the time of entry into the source state of the transition triggered by the event. In that case, the time event occurrence is generated only if the state machine is still in that state when the deadline expires.

## Notation

If no starting point is indicated, then it is the time since the entry to the current state.

### 15.3.14 Transition (from BehaviorStateMachines)

#### Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 93
- “RedefinableElement (from Kernel)” on page 125

## Description

A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.

## Attributes

- kind: TransitionKind [1] See definition of TransitionKind.

## Associations

- trigger: Trigger[0..\*] Specifies the triggers that may fire the transition.
- guard: Constraint[0..1] A guard is a constraint that provides a fine-grained control over the firing of the transition. The guard is evaluated when an event occurrence is dispatched by the state machine. If the guard is true at that time, the transition may be enabled, otherwise, it is disabled. Guards should be pure expressions without side effects. Guard expressions with side effects are ill formed.
- effect: Behavior[0..1] Specifies an optional behavior to be performed when the transition fires.
- source: Vertex[1] Designates the originating vertex (state or pseudostate) of the transition.
- target: Vertex[1] Designates the target vertex that is reached when the transition is taken.
- replacedTransition: Transition[0..1] The transition of which this is a replacement.
- /redefinitionContext: Classifier[1] References the classifier in which context this element may be redefined.
- container [1] Designates the region that owns this transition. (Subsets *Namespace.namespace*)

## Constraints

- [1] A fork segment must not have guards or triggers.  
(source.ocllsKindOf(Pseudostate) **and** source.kind = #fork) **implies** (guard->isEmpty() **and** trigger->isEmpty())
- [2] A join segment must not have guards or triggers.  
(target.ocllsKindOf(Pseudostate) **and** target.kind = #join) **implies** (guard->isEmpty() **and** trigger->isEmpty())
- [3] A fork segment must always target a state.  
(source.ocllsKindOf(Pseudostate) **and** source.kind = #fork) **implies** (target.ocllsKindOf(State))
- [4] A join segment must always originate from a state.  
(target.ocllsKindOf(Pseudostate) **and** target.kind = #join) **implies** (source.ocllsKindOf(State))
- [5] Transitions outgoing pseudostates may not have a trigger.  
source.ocllsKindOf(Pseudostate) **and**  
((source.kind <> #junction) **and** (source.kind <> #join) **and** (source.kind <> #initial)) **implies** trigger->isEmpty()
- [6] An initial transition at the topmost level (region of a statemachine) either has no trigger or it has a trigger with the stereotype "create."  
self.source.ocllsKindOf(Pseudostate) **implies**  
(self.source.ocllAsType(Pseudostate).kind = #initial) **implies**  
(self.source.container = self.stateMachine.top) **implies**  
((self.trigger->isEmpty() **or**  
(self.trigger.stereotype.name = 'create'))
- [7] In case of more than one trigger, the signatures of these must be compatible in case the parameters of the signal are assigned to local variables/attributes.
- [8] The redefinition context of a transition is the nearest containing statemachine.  
redefinitionContext =  
    let sm = containingStateMachine() in  
    if sm.context->isEmpty() or sm.general->notEmpty() then  
        sm  
    else  
        sm.context  
    endif

## Additional operations

- [1] The query isConsistentWith() specifies that a redefining transition is consistent with a redefined transition provided that the redefining transition has the following relation to the redefined transition: A redefining transition redefines all properties of the corresponding redefined transition, except the source state and the trigger.
- [2] The query containingStateMachine() returns the state machine that contains the transition either directly or transitively.  
**context** Transition::containingStateMachine() : StateMachine  
**post:** result = container.containingStateMachine()

## Semantics

### High-level transitions

Transitions originating from composite states themselves are called *high-level* or *group* transitions. If triggered, they result in exiting of all the substates of the composite state executing their exit activities starting with the innermost states in the active state configuration. Note that in terms of execution semantics, a high-level transition does not add

specialized semantics, but rather reflects the semantics of exiting a composite state. A high-level transition with a target outside the composite state will imply the execution of the exit action of the composite state, while a high-level transition with a target inside the composite state will not imply execution of the exit action of the composite state.

### *Compound transitions*

A *compound transition* is a derived semantic concept, represents a “semantically complete” path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states. The transition execution semantics described below refer to compound transitions.

In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates that define path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). For self-transitions, the same state acts as both the source and the destination set. A (simple) transition connecting two states is therefore a special common case of a compound transition.

The tail of a compound transition may have multiple transitions originating from a set of mutually orthogonal regions that are joined by a join point.

The head of a compound transition may have multiple transitions originating from a fork pseudostate targeted to a set of mutually orthogonal regions.

In a compound transition multiple outgoing transitions may emanate from a common *junction* point. In that case, only one of the outgoing transitions whose guard is true is taken. If multiple transitions have guards that are true, a transition from this set is chosen. The algorithm for selecting such a transition is not specified. Note that in this case, the guards are evaluated before the compound transition is taken.

In a compound transition where multiple outgoing transitions emanate from a common *choice* point, the outgoing transition whose guard is true *at the time the choice point is reached*, will be taken. If multiple transitions have guards that are true, one transition from this set is chosen. The algorithm for selecting this transition is not specified. If no guards are true after the choice point has been reached, the model is ill formed.

### *Internal transitions*

An internal transition executes without exiting or re-entering the state in which it is defined. This is true even if the state machine is in a nested state within this state.

### *Completion transitions and completion events*

A *completion transition* is a transition originating from a state or an exit point but which does not have an explicit trigger, although it may have a guard defined. A completion transition is implicitly triggered by a completion event. In case of a leaf state, a completion event is generated once the entry actions and the internal activities (“do” activities) have been completed. If no actions or activities exist, the completion event is generated upon entering the state. If the state is a composite state or a submachine state, a completion event is generated if either the submachine or the contained region has reached a final state and the state’s internal activities have been completed. This event is the implicit trigger for a completion transition. The completion event is dispatched before any other events in the pool and has no associated parameters. For instance, a completion transition emanating from an orthogonal composite state will be taken automatically as soon as all the orthogonal regions have reached their final state.

If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions.

### *Enabled (compound) transitions*

A transition is *enabled* if and only if:



- All of its source states are in the active state configuration.
- One of the triggers of the transition is satisfied by the event (type) of the current occurrence. An event *satisfies* a trigger if it matches the event specified by the trigger. In case of signal events, since signals are generalized concepts, a signal event satisfies a signal event associated with the same signal or a generalization thereof.
- If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice point in which all guard conditions are true (transitions without guards are treated as if their guards are always true).

Since more than one transition may be enabled by the same event, being enabled is a necessary but not sufficient condition for the firing of a transition.

### Guards

In a simple transition with a guard, the guard is evaluated before the transition is triggered.

In compound transitions involving multiple guards, all guards are evaluated before a transition is triggered, unless there are choice points along one or more of the paths. The order in which the guards are evaluated is not defined.

If there are choice points in a compound transition, only guards that precede the choice point are evaluated according to the above rule. Guards downstream of a choice point are evaluated if and when the choice point is reached (using the same rule as above). In other words, for guard evaluation, a choice point has the same effect as a state.

Guards should not include expressions causing side effects. Models that violate this are considered ill formed.

### Transition execution sequence

Every transition, except for internal and local transitions, causes exiting of a source state, and entering of the target state. These two states, which may be composite, are designated as the *main source* and the *main target* of a transition.

The *least common ancestor* (LCA) state of a (compound) transition is a region or an orthogonal state that is the LCA of the source and target states of the (compound) transition. The LCA operation is an operation defined for the `StateMachine` class.

If the LCA is a Region, then the main source is a direct subvertex of the region that contains the source states, and the main target is the subvertex of the region that contains the target states. In the case where the LCA is an orthogonal state, the main source and the main target are both represented by the orthogonal state itself. The reason is that a transition crossing regions of an orthogonal state forces exit from the entire orthogonal state and re-entering of all of its regions.

### Examples

- The common simple case: A transition *t* between two simple states *s1* and *s2*, in a composite state. Here the least common ancestor of *t* is *s*, the main source is *s1*, and the main target is *s2*.

Note that a transition from one region to another in the same immediate enclosing composite state is not allowed: the two regions must be part of two different composite states. Here least common ancestor of *t* is *s*, the main source is *s* and the main target is *s*, since *s* is an orthogonal state as specified above.

Once a transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source state is properly exited.
- Behaviors are executed in sequence following their linear order along the segments of the transition: The closer the behavior to the source state, the earlier it is executed.

- If a choice point is encountered, the guards following that choice point are evaluated dynamically and a path whose guards are true is selected.
- The main target state is properly entered.

### Transition redefinition

A transition of an extended state machine may in the state machine extension be redefined. A redefinition transition redefines all properties of the corresponding replaced transition in the extended state machine, except the source state and the trigger.

### Notation

The default notation for a transition is defined by the following BNF expression:

$$\langle \text{transition} \rangle ::= \langle \text{trigger} \rangle [', ' \langle \text{trigger} \rangle]^* [ '[' \langle \text{guard-constraint} \rangle ' ] ' ] [ '/' \langle \text{activity-expression} \rangle ]$$

However, relative to its use for signal events (see “SignalEvent (from Communications)” on page 435) and change events (see “ChangeEvent (from Communications)” on page 422), the *<assignment-specification>* when used in transitions is extended as follows:

$$\begin{aligned} \langle \text{assignment-specification} \rangle &::= \langle \text{attr-spec} \rangle [', ' \langle \text{attr-spec} \rangle]^* \\ \langle \text{attr-spec} \rangle &::= \langle \text{attr-name} \rangle [ ':' \langle \text{type-name} \rangle ] \end{aligned}$$

Note that *<attr-name>* is the name of an attribute to which the corresponding parameter value of the event is assigned. If a *<type-name>* is included with the attribute name, then it represents an implicit declaration of a local attribute of that type in the context of the effect activity to which the corresponding parameter value of the event is assigned.

The *<guard-constraint>* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the context object. The guard constraint may also involve tests of orthogonal states of the current state 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 and regions that contain them, yielding pathnames of the form “(RegionOrState1::RegionOrState2::State3.” This may be used in case the same state name occurs in different composite state regions.

The *behavior-expression* is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the context object and the parameters of the triggering event, or any other features visible in its scope. The behavior 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.

Both in the behavior-expression and in actions of the effect transition specified graphically, the values of the signal instance (in case of a signal trigger) are denoted by *signal-name* ‘.’ *attribute-name* in case of just one signal trigger, and by ‘msg.’ *attr-name* in case of more than one trigger.

Internal transitions are specified in a special compartment of the source state, see Figure 15.32.

### Presentation options

The triggers and the subsequent effect of a transition may be notated either textually or as a presentation option, using graphical symbols on a transition. This section describes the graphical presentation option.

The graphical presentation of triggers and effect of a transition consists of one or more graphical symbols attached to a line connecting the symbols for source and target of the transition each representing a trigger or an action of the transition effect. The action symbols split one logical transition into several graphical line segments with an arrowhead on one end (see Figure 15.44).

The sequence of symbols may consist of a single trigger symbol (or none), followed by a sequence of zero or more *action* symbols. The trigger symbol maps to the set of Triggers. The sequence of action symbols maps to the Behavior representing the effect of the transition. Each action symbol is mapped to an action contained within the single SequenceNode comprising an Activity that is the effect Behavior. The SequenceNode orders the actions according to their graphical order on the transition path.

All line segments connecting the symbols representing source and target of the transition as well as all action symbols represent a single transition, and map to a single behavior. This behavior owns the actions that are represented by the action symbols.

### *Signal receipt*

The trigger symbol is shown as a five-pointed polygon that looks like a rectangle with a triangular notch in one of its sides (either one). It represents the trigger of the transition. The textual trigger specification is denoted within the symbol. If the transition owns a guard, the guard is also described within the signal receipt icon. The textual notation:

`<trigger> ['<trigger>']* ['<guard> ']`

The trigger symbol is always first in the path of symbols and a transition path can only have at most one such symbol.

### *Signal sending*

Signal sending is a common action that has a special notation described in SendSignalAction. The actual parameters of the signal are shown within the symbol. On a given path, the signal sending symbol must follow the trigger symbol if the latter exists.

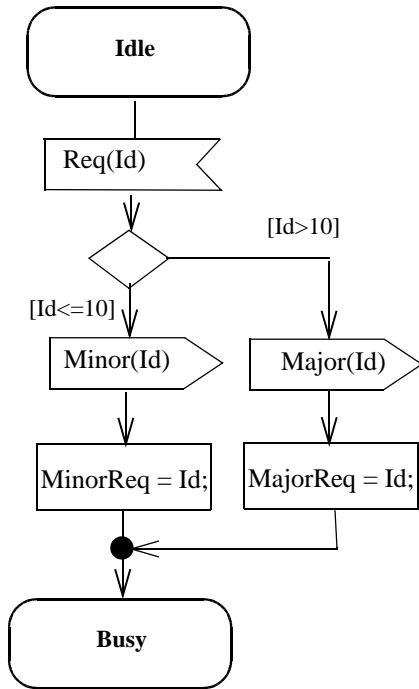
The signal sending symbol maps to a SendSignalAction. If a tool does not use the actions package, the details of the sent signal may be captured within the body of the behavior instead of the SendSignalAction instance. It is possible to have multiple signal sending nodes on a transition path.

### *Other actions*

An action symbol is a rectangle that contains a textual representation of the action represented by this symbol. Alternatively, the graphical notation defined for this action, if any, may be used instead. The action symbol must follow the signal receipt symbol if one exists for that transition. The action sequence symbol is mapped to an opaque action, or to a sequence node containing instances of actions, depending on whether it represents one or more actions, respectively.

Note that this presentation option combines only the above symbols into a single transition. Choice and junction symbols (see Figure 15.44) for example, are not mapped to actions on transition but to choice and junction pseudo-states.

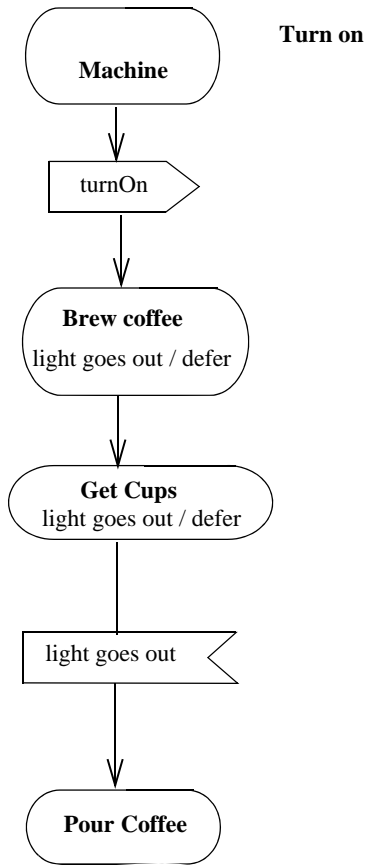
Therefore, Figure 15.44 shows these transitions: one from the idle state to the choice symbol, one for each of the branches of the choice through the junction symbol, and one from the junction pseudostate to the busy state.



**Figure 15.44 - Symbols for Signal Receipt, Sending and Actions on transition**

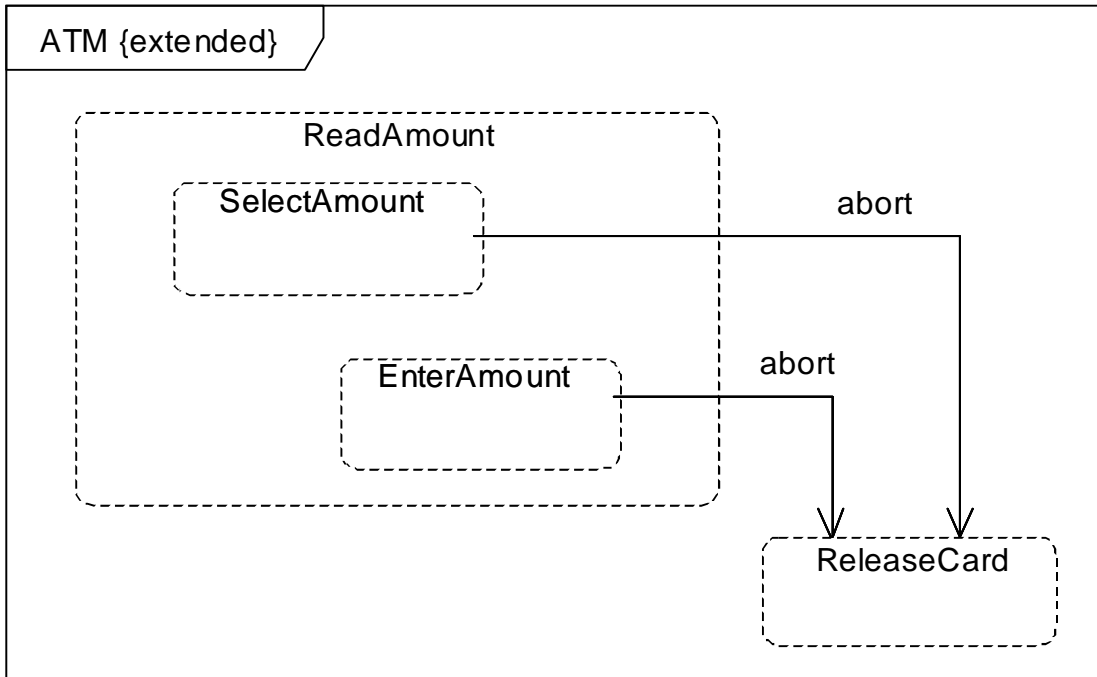
#### *Deferred triggers*

A deferrable trigger 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 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 pool.



**Figure 15.45 - Deferred Trigger Notation**

Figure 15.46 shows an example of adding transitions in a specialized state machine.



**Figure 15.46 - Adding Transitions**

### Example

Transition with guard constraint and transition string:

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

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

### Changes from previous UML

- Transition redefinition has been added, in order to express if a transition of a general state machine can be redefined or not.
- Along with this, an association to the redefined transition has been added.
- Guard has been replaced by a Constraint.

### 15.3.15 TransitionKind (from BehaviorStateMachines)

TransitionKind is an enumeration type.

## Generalizations

None

## Description

TransitionKind is an enumeration of the following literal values:

- external
- internal
- local

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

- [1] The source state of a transition with transition kind *local* must be a composite state.
- [2] The source state of a transition with transition kind *external* must be a composite state.

## Semantics

- kind=internal implies that the transition, if triggered, occur without exiting or entering the source state. Thus, it does not cause a state change. This means that the entry or exit condition of the source state will not be invoked. An internal transition can be taken even if the state machine is in one or more regions nested within this state.
- kind=local implies that the transition, if triggered, will not exit the composite (source) state, but it will apply to any state within the composite stat, and these will be exited and entered.
- kind=external implies that the transition, if triggered, will exit the composite (source) state.

## Notation

- Transitions of kind *local* will be on the inside of the frame of the composite state, leaving the border of the composite state and end at a vertex *inside* the composite state. Alternatively a transition of kind local can be shown as a transition leaving a state symbol containing the text “\*.” The transition is then considered to belong to the enclosing composite state.
- Transitions of kind *external* will leave the border of the composite state and end at either a vertex *outside* the composite state or the composite state itself.

## Changes from previous UML

The semantics implied by *local* is new.

### 15.3.16 Vertex (from BehaviorStateMachines)

## Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 93

## Description

A vertex is an abstraction of a node in a state machine graph. In general, it can be the source or destination of any number of transitions.

## Attributes

No additional attributes

## Associations

- outgoing: Transition[1..\*] Specifies the transitions departing from this vertex.
- incoming: Transition[1..\*] Specifies the transitions entering this vertex.
- container: Region[0..1] The region that contains this vertex. {Subsets *Element::owner*}

## Additional operations

[1] The operation containingStateMachine() returns the state machine in which this Vertex is defined.

**context** Region::containingStateMachine() : StateMachine

**post:** result = if not container->isEmpty() then

-- the container is a region

container.containingStateMachine()

else if (oclIsKindOf(Pseudostate)) then

-- entry or exit point?

if (kind = #entryPoint) or (kind = #exitPoint) then

stateMachine

else if (oclIsKindOf(ConnectionPointReference)) then

state.containingStateMachine() -- no other valid cases possible

## 15.4 Diagrams

State machine diagrams specify state machines. This chapter outlines the graphic elements that may be shown in state machine diagrams, and provides cross references where detailed information about the semantics and concrete notation for each element can be found. It also furnishes examples that illustrate how the graphic elements can be assembled into diagrams.

### Graphic Nodes

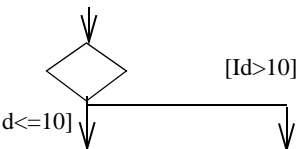
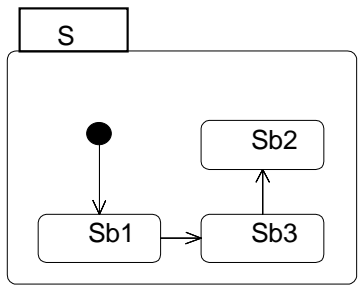






The graphic nodes that can be included in state machine diagrams are shown in Table 15.1.

**Table 15.1 - Graphic nodes included in state machine diagrams**


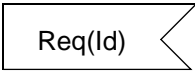
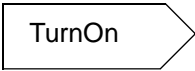
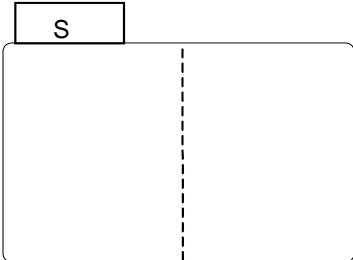
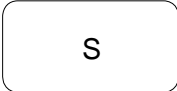
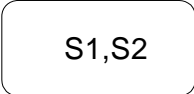
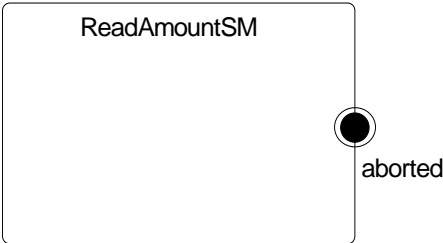
| Node Type | Notation                   | Reference                                                  |
|-----------|----------------------------|------------------------------------------------------------|
| Action    | <div>MinorReq := Id;</div> | See “Transition (from BehaviorStateMachines)” on page 553. |




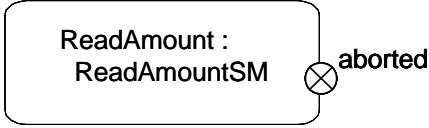
**Table15.1 - Graphic nodes included in state machine diagrams**

| Node Type                     | Notation                                                                            | Reference                                                                    |
|-------------------------------|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Choice pseudo state           |    | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| Composite state               |    | See “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 531. |
| Entry point                   |  | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| Exit point                    |  | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| Final state                   |  | See “FinalState (from BehaviorStateMachines)” on page 513.                   |
| History, Deep Pseudo state    |  | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| History, Shallow pseudo state |  | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| Initial pseudo state          |  | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |

**Table15.1 - Graphic nodes included in state machine diagrams**

| Node Type             | Notation                                                                            | Reference                                                                    |
|-----------------------|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Junction pseudo state |    | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| Receive signal action |    | See “Transition (from BehaviorStateMachines)” on page 553.                   |
| Send signal action    |    | See “Transition (from BehaviorStateMachines)” on page 553.                   |
| Region                |   | See “Region (from BehaviorStateMachines)” on page 529.                       |
| Simple state          |  | See “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 531. |
| State list            |  | See “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 531. |
| State Machine         |  | See “StateMachine (from BehaviorStateMachines)” on page 545.                 |

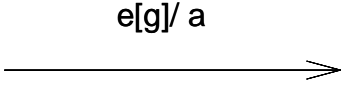
**Table15.1 - Graphic nodes included in state machine diagrams**

| Node Type        | Notation                                                                          | Reference                                                                    |
|------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| Terminate node   |  | See “Pseudostate (from BehaviorStateMachines)” on page 522.                  |
| Submachine state |  | See “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 531. |

### Graphic Paths

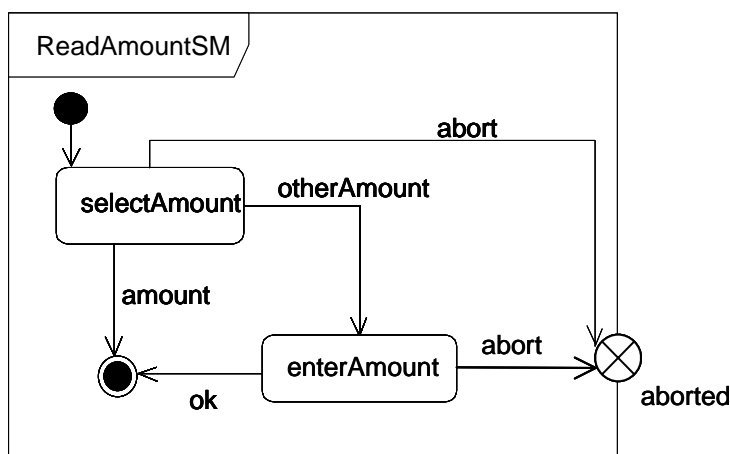
The graphic paths that can be included in state machine diagrams are shown in Table15.2.

**Table15.2 - Graphic paths included in state machine diagrams**

| Path Type  | Notation                                                                           | Reference                                                  |
|------------|------------------------------------------------------------------------------------|------------------------------------------------------------|
| Transition |  | See “Transition (from BehaviorStateMachines)” on page 553. |

### Examples

The following are examples of state machine diagrams.



**Figure 15.47 - State machine with exit point definition**

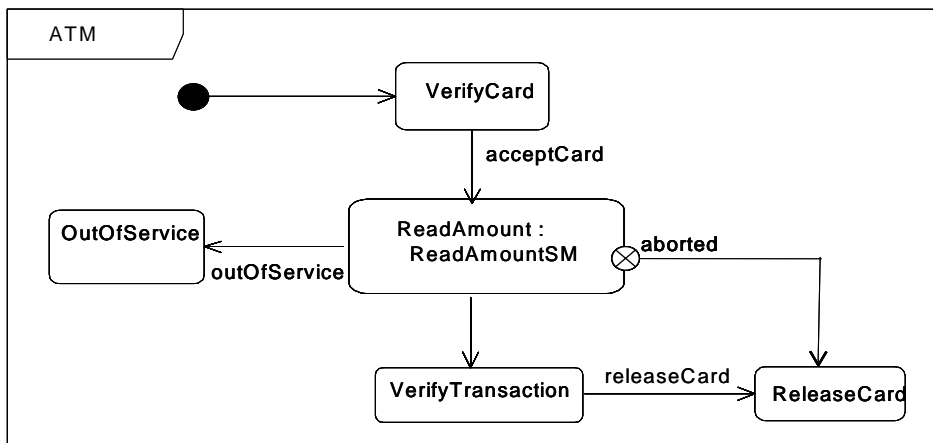


Figure 15.48 - SubmachineState with usage of exit point



## 16 Use Cases

### 16.1 Overview

Use cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do. The key concepts associated with use cases are *actors*, *use cases*, and the *subject*. The subject is the system under consideration to which the use cases apply. The users and any other systems that may interact with the subject are represented as actors. Actors always model entities that are outside the system. The required behavior of the subject is specified by one or more use cases, which are defined according to the needs of actors.

Strictly speaking, the term “use case” refers to a use case type. An instance of a use case refers to an occurrence of the emergent behavior that conforms to the corresponding use case type. Such instances are often described by interaction specifications.

Use cases, actors, and systems are described using use case diagrams.

### 16.2 Abstract syntax

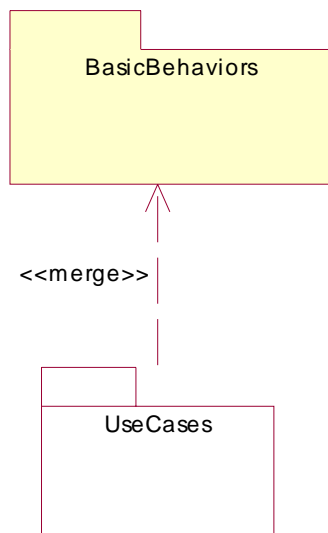


Figure 16.1 - Dependencies of the UseCases package

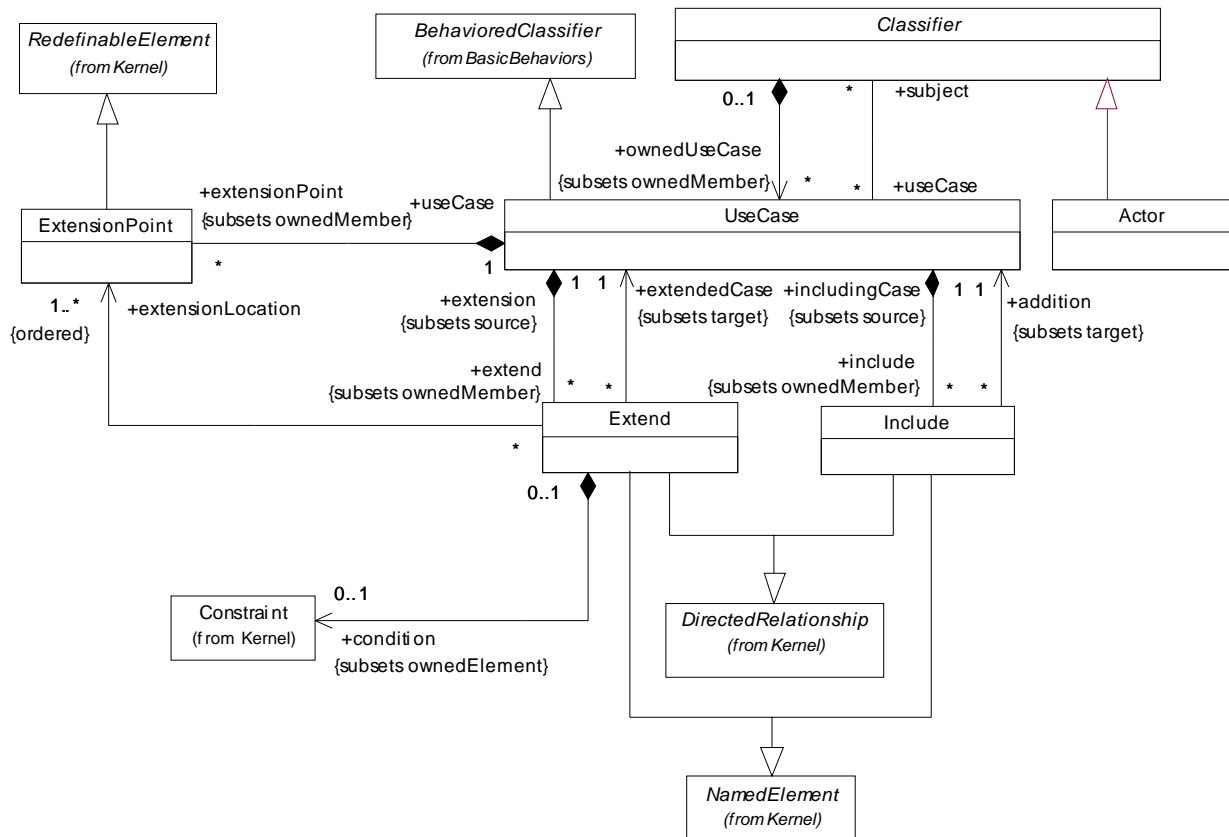


Figure 16.2 - The concepts used for modeling use cases

## 16.3 Class Descriptions

### 16.3.1 Actor (from UseCases)

An actor specifies a role played by a user or any other system that interacts with the subject. (The term “role” is used informally here and does not necessarily imply the technical definition of that term found elsewhere in this specification.)

#### Generalizations

- “Classifier (from UseCases)” on page 572

#### Description

An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is *external* to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., “role”) of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances.

Since an actor is external to the subject, it is typically defined in the same classifier or package that incorporates the subject classifier.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

[1] An actor can only have associations to use cases, components, and classes. Furthermore these associations must be binary.

```
self.ownedAttribute->forAll ( a |  
    (a.association->notEmpty()) implies  
        ((a.association.memberEnd.size() = 2) and  
        (a.opposite.class.oclIsKindOf(UseCase) or  
        (a.opposite.class.oclIsKindOf(Class) and not a.opposite.class.oclIsKindOf(Behavior))))
```

[2] An actor must have a name.

```
name->notEmpty()
```

### Semantics

Actors model entities external to the subject. When an external entity interacts with the subject, it plays the role of a specific actor.

When an actor has an association to a use case with a multiplicity that is greater than one at the use case end, it means that a given actor can be involved in multiple use cases of that type. The specific nature of this multiple involvement depends on the case on hand and is not defined in this specification. Thus, an actor may initiate multiple use cases in parallel (concurrently) or they may be mutually exclusive in time. For example, a computer user may activate a given software application multiple times concurrently or at different points in time.

### Notation

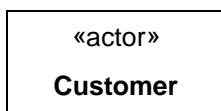
An actor is represented by “stick man” icon with the name of the actor in the vicinity (usually above or below) the icon.



**Customer**

### Presentation Options

An actor may also be shown as a class rectangle with the keyword «actor», with the usual notation for all compartments.





Other icons that convey the kind of actor may also be used to denote an actor, such as using a separate icon for non-human actors.



## Style Guidelines

Actor names should follow the capitalization and punctuation guidelines used for classes in the model. The names of abstract actors should be shown in italics.

## Changes from previous UML

There are no changes to the Actor concept except for the addition of a constraint that requires that all actors must have names.

### 16.3.2 Classifier (from UseCases)

#### Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 48 (*merge increment*)

#### Description

Extends a classifier with the capability to own use cases. Although the owning classifier typically represents the subject to which the owned use cases apply, this is not necessarily the case. In principle, the same use case can be applied to multiple subjects, as identified by the *subject* association role of a UseCase (see “UseCase (from UseCases)” on page 578).

#### Attributes

No additional attributes

#### Associations

- ownedUseCase: UseCase[\*]      References the use cases owned by this classifier.  
(Subsets *Namespace.ownedMember*)
- useCase : UseCase [\*]      The set of use cases for which this Classifier is the subject.

#### Constraints

No additional constraints

#### Semantics

See “UseCase (from UseCases)” on page 578.

## Notation

The nesting (owning) of a use case by a classifier is represented using the standard notation for nested classifiers.



## Rationale

This extension to the Classifier concept was added to allow classifiers in general to own use cases.

## Changes from previous UML

No changes

### 16.3.3 Extend (from UseCases)

A relationship from an extending use case to an extended use case that specifies how and when the behavior defined in the extending use case can be inserted into the behavior defined in the extended use case.

## Generalizations

- “DirectedRelationship (from Kernel)” on page 59

## Description

This relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case. Note, however, that the extended use case is defined independently of the extending use case and is meaningful independently of the extending use case. On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions.

Note that the same extending use case can extend more than one use case. Furthermore, an extending use case may itself be extended.

It is a kind of DirectedRelationship, such that the source is the extending use case and the destination is the extended use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The extend relationship itself is owned by the extending use case.

## Attributes

No additional attributes

## Associations

- extendedCase : UseCase [1]      References the use case that is being extended.  
(Specializes *DirectedRelationship.target*)

- **extension : UseCase [1]**      References the use case that represents the extension and owns the extend relationship. (Specializes *DirectedRelationship.source*)
- **condition : Constraint [0..1]**      References the condition that must hold when the first extension point is reached for the extension to take place. If no constraint is associated with the extend relationship, the extension is unconditional. (Specializes *Element.ownedElement*)
- **extensionLocation: ExtensionPoint [1..\*]**      An ordered list of extension points belonging to the extended use case, specifying where the respective behavioral fragments of the extending use case are to be inserted. The first fragment in the extending use case is associated with the first extension point in the list, the second fragment with the second point, and so on. (Note that, in most practical cases, the extending use case has just a single behavior fragment, so that the list of extension points is trivial.)

## Constraints

[1] The extension points referenced by the extend relationship must belong to the use case that is being extended.

extensionLocation->forAll (xp | extendedCase.extensionPoint->includes(xp))

## Semantics

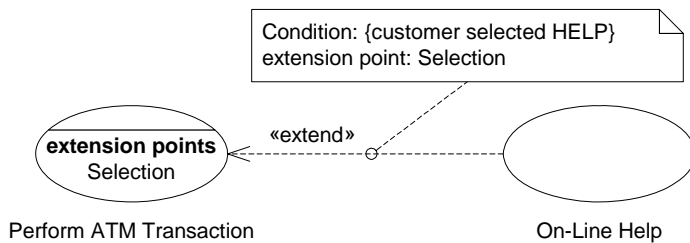
The concept of an “extension location” is intentionally left underspecified because use cases are typically specified in various idiosyncratic formats such as natural language, tables, trees, etc. Therefore, it is not easy to capture its structure accurately or generally by a formal model. The intuition behind the notion of extension location is best explained through the example of a textually described use case: Usually, a use case with extension points consists of a set of finer-grained behavioral fragment descriptions, which are most often executed in sequence. This segmented structuring of the use case text allows the original behavioral description to be extended by merging in supplementary behavioral fragment descriptions at the appropriate insertion points between the original fragments (extension points). Thus, an extending use case typically consists of one or more behavior fragment descriptions that are to be inserted into the appropriate spots of the extended use case. An extension location, therefore, is a specification of all the various (extension) points in a use case where supplementary behavioral increments can be merged.

If the condition of the extension is true at the time the first extension point is reached during the execution of the extended use case, then all of the appropriate behavior fragments of the extending use case will also be executed. If the condition is false, the extension does not occur. The individual fragments are executed as the corresponding extension points of the extending use case are reached. Once a given fragment is completed, execution continues with the behavior of the extended use case following the extension point. Note that even though there are multiple use cases involved, there is just a single behavior execution.

## Notation

An extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the use case providing the extension to the base use case. The arrow is labeled with the keyword «extend». The condition of the relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship. (See Figure 16.3.)

## Examples



**Figure 16.3 - Example of an extend relationship between use cases**

In the use case diagram above, the use case “Perform ATM Transaction” has an extension point “Selection.” This use case is extended via that extension point by the use case “On-Line Help” whenever execution of the “Perform ATM Transaction” use case occurrence is at the location referenced by the “Selection” extension point and the customer selects the HELP key. Note that the “Perform ATM Transaction” use case is defined independently of the “On-Line Help” use case.

## Rationale

This relationship is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in another use case (which is meaningful independently of the extending use case).

## Changes from previous UML

The notation for conditions has been changed such that the condition and the referenced extension points may now be included in a Note attached to the extend relationship, instead of merely being a textual comment that is located in the vicinity of the relationship.

### 16.3.4 ExtensionPoint (from UseCases)

An extension point identifies a point in the behavior of a use case where that behavior can be extended by the behavior of some other (extending) use case, as specified by an extend relationship.

## Generalizations

- “RedefinableElement (from Kernel)” on page 125

## Description

An ExtensionPoint is a feature of a use case that identifies a point where the behavior of a use case can be augmented with elements of another (extending) use case.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

- [1] An ExtensionPoint must have a name.  
self.name->notEmpty ()

## Semantics

An extension point is a reference to a location within a use case at which parts of the behavior of other use cases may be inserted. Each extension point has a unique name within a use case.

## Semantic Variation Points

The specific manner in which the location of an extension point is defined is left as a semantic variation point.

## Notation

Extension points are indicated by a text string within in the use case oval symbol or use case rectangle according to the syntax below:

*<extension point>* ::= <name> [: <explanation>]

Note that *explanation*, which is optional, may be any informal text or a more precise definition of the location in the behavior of the use case where the extension point occurs.

## Examples

See Figure 16.3 on page 575 and Figure 16.9 on page 582.

## Rationale

ExtensionPoint supports the use case extension mechanism (see “Extend (from UseCases)” on page 573).

## Changes from previous UML

In 1.x, ExtensionPoint was modeled as a kind of ModelElement, which due to a multiplicity constraint, was always associated with a specific use case. This relationship is now modeled as an owned feature of a use case. Semantically, this is equivalent and the change will not be visible to users.

ExtensionPoints in 1.x had an attribute called *location*, which was a kind of LocationReference. Since the latter had no specific semantics it was relegated to a semantic variation point. When converting to UML 2.0, models in which ExtensionPoints had a *location* attribute defined, the contents of the attribute should be included in a note attached to the ExtensionPoint.

### 16.3.5 Include (from UseCases)

An include relationship defines that a use case contains the behavior defined in another use case.

## Generalizations

- “DirectedRelationship (from Kernel)” on page 59

## Description

Include is a *DirectedRelationship* between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case. It is also a kind of *NamedElement* so that it can have a name in the context of its owning use case. The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case.

Note that the included use case is not optional, and is always required for the including use case to execute correctly.

## Attributes

No additional attributes

## Associations

- addition : UseCase [1]      References the use case that is to be included. (Specializes *DirectedRelationship.target*)
- including Case : UseCase [1]      References the use case that will include the addition and owns the include relationship. (Specializes *DirectedRelationship.source*)

## Constraints

No additional constraints

## Semantics

An include relationship between two use cases means that the behavior defined in the including use case is included in the behavior of the base use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is then extracted to a separate use case, to be included by all the base use cases having this part in common. Since the primary use of the include relationship is for reuse of common parts, what is left in a base use case is usually not complete in itself but dependent on the included parts to be meaningful. This is reflected in the direction of the relationship, indicating that the base use case depends on the addition but not vice versa.

Execution of the included use case is analogous to a subroutine call. All of the behavior of the included use case is executed at a single location in the included use case before execution of the including use case is resumed.

## Notation

An include relationship between use cases is shown by a dashed arrow with an open arrowhead from the base use case to the included use case. The arrow is labeled with the keyword «include». (See Figure 16.4.)

## Examples

A use case “Withdraw” includes an independently defined use case “Card Identification.”



Figure 16.4 - Example of the Include relationship

## Rationale

The Include relationship allows hierarchical composition of use cases as well as reuse of use cases.

## Changes from previous UML

There are no changes to the semantics or notation of the Include relationship relative to UML 1.x.

### 16.3.6 UseCase (from UseCases)

A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.

## Generalizations

- “BehavioredClassifier (from BasicBehaviors, Communications)” on page 419

## Description

A UseCase is a kind of behaviored classifier that represents a declaration of an offered behavior. Each use case specifies some behavior, possibly including variants, that the subject can perform in collaboration with one or more actors. Use cases define the offered behavior of the subject without reference to its internal structure. These behaviors, involving interactions between the actor and the subject, may result in changes to the state of the subject and communications with its environment. A use case can include possible variations of its basic behavior, including exceptional behavior and error handling.

The subject of a use case could be a physical system or any other element that may have behavior, such as a component, subsystem, or class. Each use case specifies a unit of useful functionality that the subject provides to its users (i.e., a specific way of interacting with the subject). This functionality, which is initiated by an actor, must always be completed for the use case to complete. It is deemed complete if, after its execution, the subject will be in a state in which no further inputs or actions are expected and the use case can be initiated again or in an error state.

Use cases can be used both for specification of the (external) requirements on a subject and for the specification of the functionality offered by a subject. Moreover, the use cases also state the requirements the specified subject poses on its environment by defining how they should interact with the subject so that it will be able to perform its services.

The behavior of a use case can be described by a specification that is some kind of Behavior (through its ownedBehavior relationship), such as interactions, activities, and state machines, or by pre-conditions and post-conditions as well as by natural language text where appropriate. It may also be described indirectly through a Collaboration that uses the use case and its actors as the classifiers that type its parts. Which of these techniques to use depends on the nature of the use case behavior as well as on the intended reader. These descriptions can be combined. An example of a use case with an associated state machine description is shown in Figure 16.6.

## Attributes

No additional attributes

## Associations

- subject : Classifier[\*]      References the subjects to which this use case applies. The subject or its parts realize all the use cases that apply to this subject. Use cases need not be attached to any specific subject, however. The subject may, but need not, own the use cases that apply to it.

- `include : Include[*]`      References the Include relationships owned by this use case.  
(Specializes *Classifier.feature* and *Namespace.ownedMember*)
- `extend : Extend[*]`      References the Extend relationships owned by this use case.  
(Specializes *Classifier.feature* and *Namespace.ownedMember*)
- `extensionPoint: ExtensionPoint[*]`      References the ExtensionPoints owned by the use case.  
(Specializes *Namespace.ownedMember*)

## Constraints

- [1] A UseCase must have a name.  
`self.name -> notEmpty ()`
- [2] UseCases can only be involved in binary Associations.
- [3] UseCases cannot have Associations to UseCases specifying the same subject.
- [4] A use case cannot include use cases that directly or indirectly include it.  
`not self.allIncludedUseCases()->includes(self)`

## Additional Operations

- [1] The query `allIncludedUseCases()` returns the transitive closure of all use cases (directly or indirectly) included by this use case.  
`UseCase::allIncludedUseCases() : Set(UseCase)`  
`allIncludedUseCases = self.include->union(self.include->collect(in | in.allIncludedUseCases()))`

## Semantics

An execution of a use case is an occurrence of emergent behavior.

Every instance of a classifier realizing a use case must behave in the manner described by the use case.

Use cases may have associated actors, which describes how an instance of the classifier realizing the use case and a user playing one of the roles of the actor interact. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the subject. It is not possible to state anything about the internal behavior of the actor apart from its communications with the subject.

When a use case has an association to an actor with a multiplicity that is greater than one at the actor end, it means that more than one actor instance is involved in initiating the use case. The manner in which multiple actors participate in the use case depends on the specific situation on hand and is not defined in this specification. For instance, a particular use case might require simultaneous (concurrent) action by two separate actors (e.g., in launching a nuclear missile) or it might require complementary and successive actions by the actors (e.g., one actor starting something and the other one stopping it).

## Notation

A use case is shown as an ellipse, either containing the name of the use case or with the name of the use case placed below the ellipse. An optional stereotype keyword may be placed above the name and a list of properties included below the name. If a subject (or system boundary) is displayed, the use case ellipse is visually located inside the system boundary rectangle. Note that this does not necessarily mean that the subject classifier owns the contained use cases, but merely that the use case applies to that classifier. For example, the use cases shown in Figure 16.5 on page 580 apply to the “ATMsystem” classifier but are owned by various packages as shown in Figure 16.7.



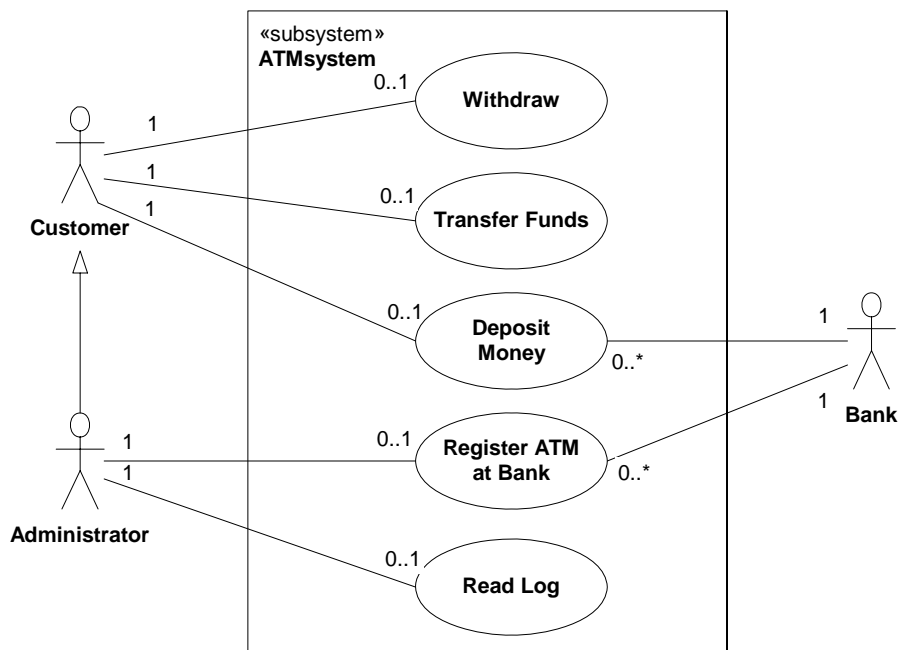


Figure 16.5 - Example of the use cases and actors for an ATM system

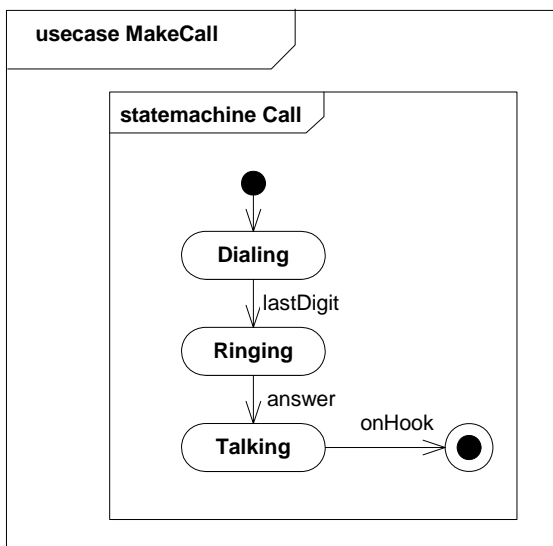
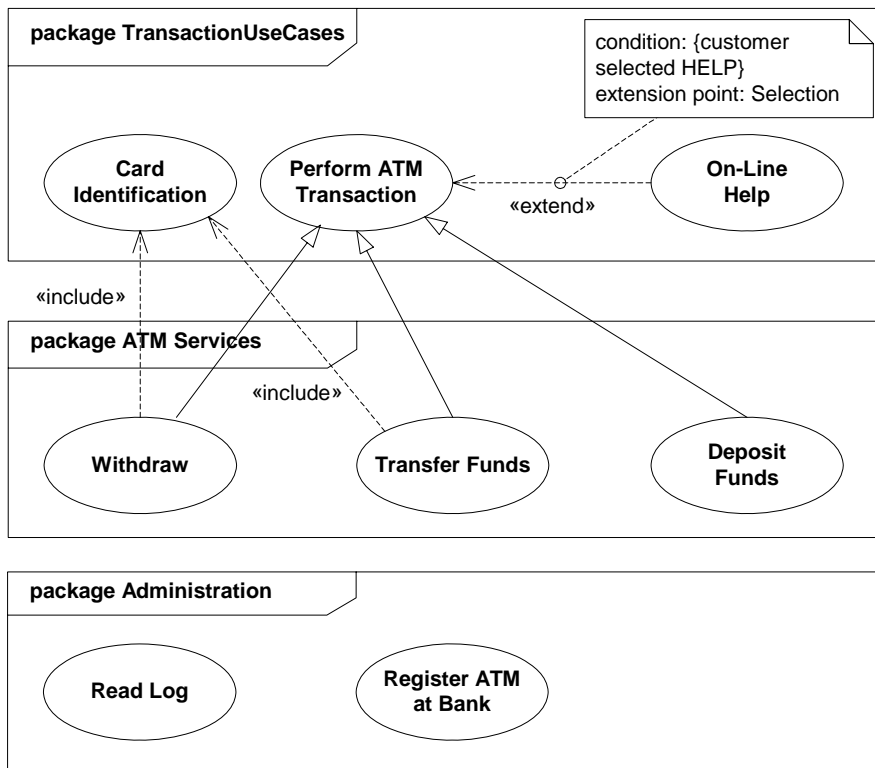


Figure 16.6 - Example of a use case with an associated state machine behavior



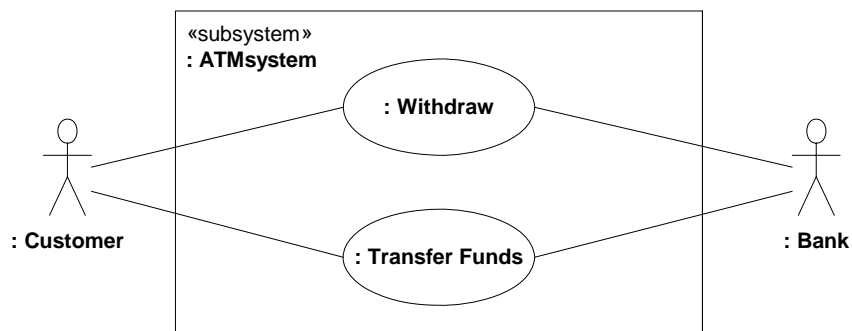
**Figure 16.7 - Example of use cases owned by various packages**

Extension points may be listed in a compartment of the use case with the heading **extension points**. The description of the locations of the extension point is given in a suitable form, usually as ordinary text, but can also be given in other forms, such as the name of a state in a state machine, an activity in an activity diagram, a precondition, or a postcondition.

Use cases may have other associations and dependencies to other classifiers (e.g., to denote input/output, events, and behaviors).

The detailed behavior defined by a use case is notated according to the chosen description technique, in a separate diagram or textual document. Operations and attributes are shown in a compartment within the use case.

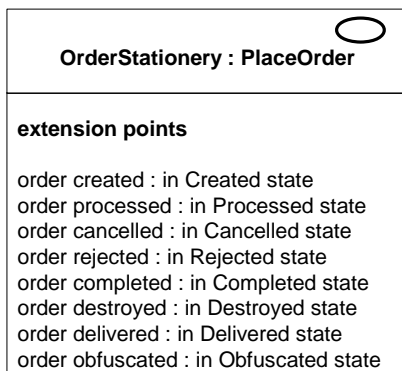
Use cases and actors may represent roles in collaborations as indicated in Figure 16.8.



**Figure 16.8 - Example of a use case for withdrawal and transfer of funds**

## Presentation Options

A use case can also be shown using the standard rectangle notation for classifiers with an ellipse icon in the upper-right-hand corner of the rectangle with optional separate list compartments for its features. This rendering is more suitable when there are a large number of extension points.



**Figure 16.9 - Example of the classifier based notation for a use case**

## Examples

See Figure 16.3 through Figure 16.9.

## Rationale

The purpose of use cases is to identify the required functionality of a system.

## Changes from previous UML

The relationship between a use case and its subject has been made explicit. Also, it is now possible for use cases to be owned by classifiers in general and not just packages.

# 16.4 Diagrams



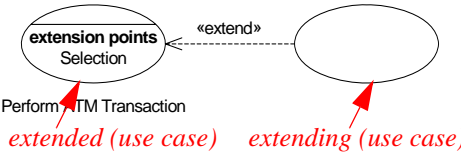
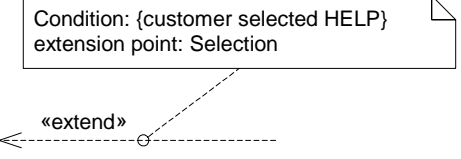

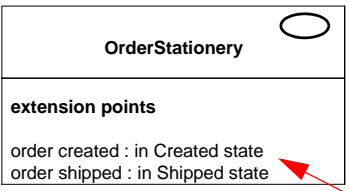
## Description

Use Case Diagrams are a specialization of Class Diagrams such that the classifiers shown are restricted to being either Actors or Use Cases.

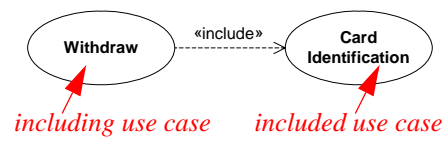

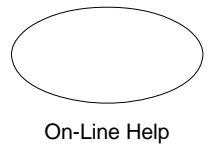
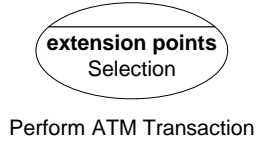

## Graphic Nodes

The graphic nodes that can be included in structural diagrams are shown in Table 16.1.

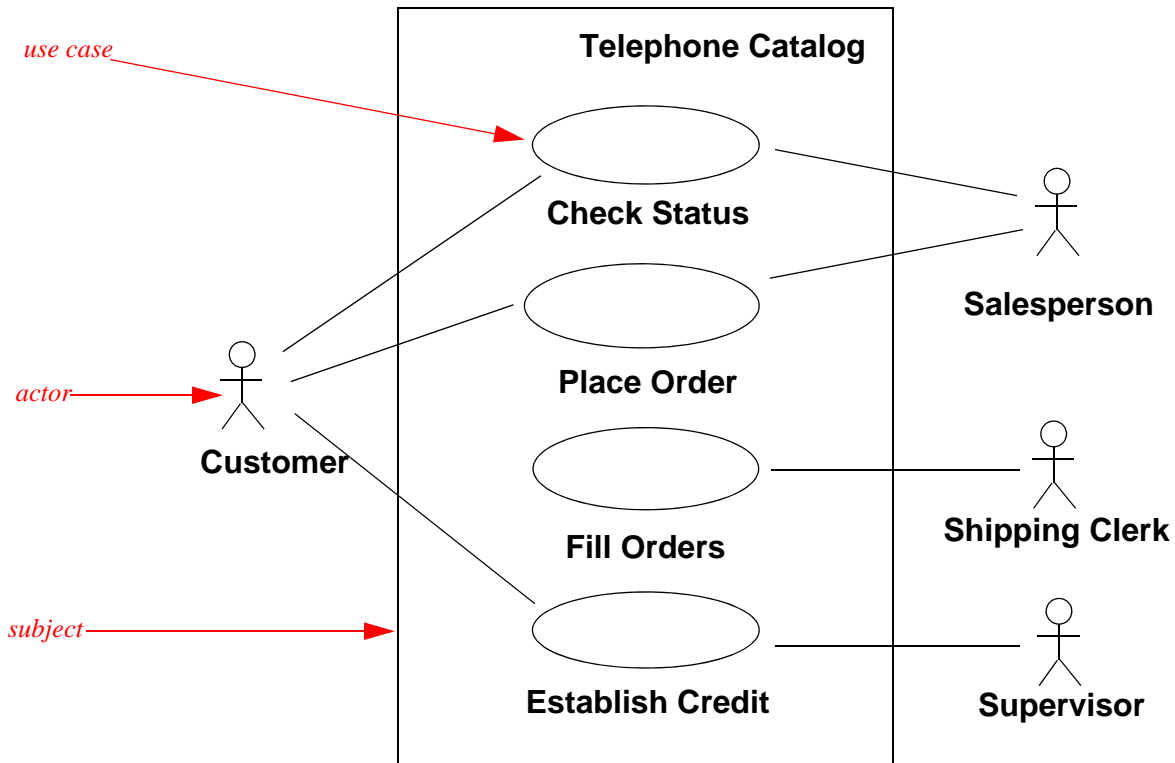
**Table 16.1 - Graphic nodes included in use case diagrams**

| Node Type                                    | Notation                                                                                                 | Reference                                         |
|----------------------------------------------|----------------------------------------------------------------------------------------------------------|---------------------------------------------------|
| Actor (default)                              |  <p><b>Customer</b></p> | See “Actor (from UseCases)” on page 570.          |
| Actor (optional user-defined icon - example) |                         |                                                   |
| Extend                                       |                        | See “Extend (from UseCases)” on page 573.         |
| Extend (with Condition)                      |                       |                                                   |
| ExtensionPoint                               |                       | See “ExtensionPoint (from UseCases)” on page 575. |
|                                              |                       |                                                   |

**Table 16.1 - Graphic nodes included in use case diagrams**

| Node Type | Notation                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Reference                                  |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| Include   |  <p>The diagram illustrates an include relationship between two use cases. On the left is an oval labeled "Withdraw", and on the right is an oval labeled "Card Identification". A dashed arrow with the label "«include»" points from "Withdraw" to "Card Identification". Below the "Withdraw" oval, a red arrow points to it with the text "including use case". Below the "Card Identification" oval, a red arrow points to it with the text "included use case".</p> | See “Include (from UseCases)” on page 576. |
| UseCase   |  <p>A standard use case represented by an oval containing the text "Withdraw".</p>                                                                                                                                                                                                                                                                                                                                                                                        | See “UseCase (from UseCases)” on page 578. |
|           |  <p>A use case represented by an oval. It has a horizontal line across its top. Below the line, the text "On-Line Help" is written. Inside the oval, below the line, is the text "On-Line Help".</p>                                                                                                                                                                                                                                                                      |                                            |
|           |  <p>A use case represented by an oval. It has a horizontal line across its top. Above the line, the text "extension points" is written. Below the line, the text "Selection" is written. Below the oval, the text "Perform ATM Transaction" is written.</p>                                                                                                                                                                                                             |                                            |
|           |  <p>A use case represented by a rectangle. Inside the rectangle, the text "OrderStationery" is written. To the right of the rectangle is a small circle.</p>                                                                                                                                                                                                                                                                                                            |                                            |

## Examples

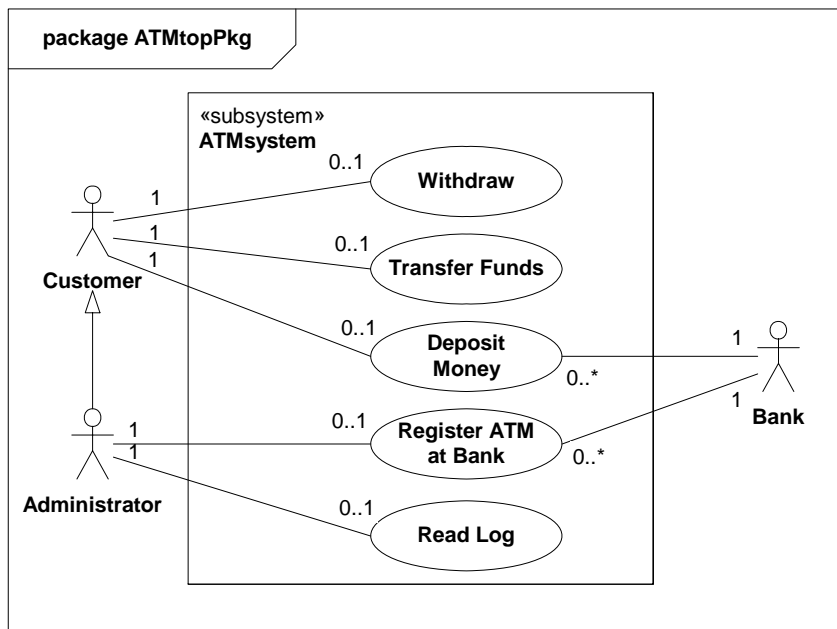


**Figure 16.10 - UseCase diagram with a rectangle representing the boundary of the subject**

The use case diagram in Figure 16.10 shows a set of use cases used by four actors of a physical system that is the subject of those use cases. The subject can be optionally represented by a rectangle as shown in this example.

Figure 16.11 illustrates a package that owns a set of use cases.

**Note –** A use case may be owned either by a package or by a classifier (typically the classifier specifying the subject).



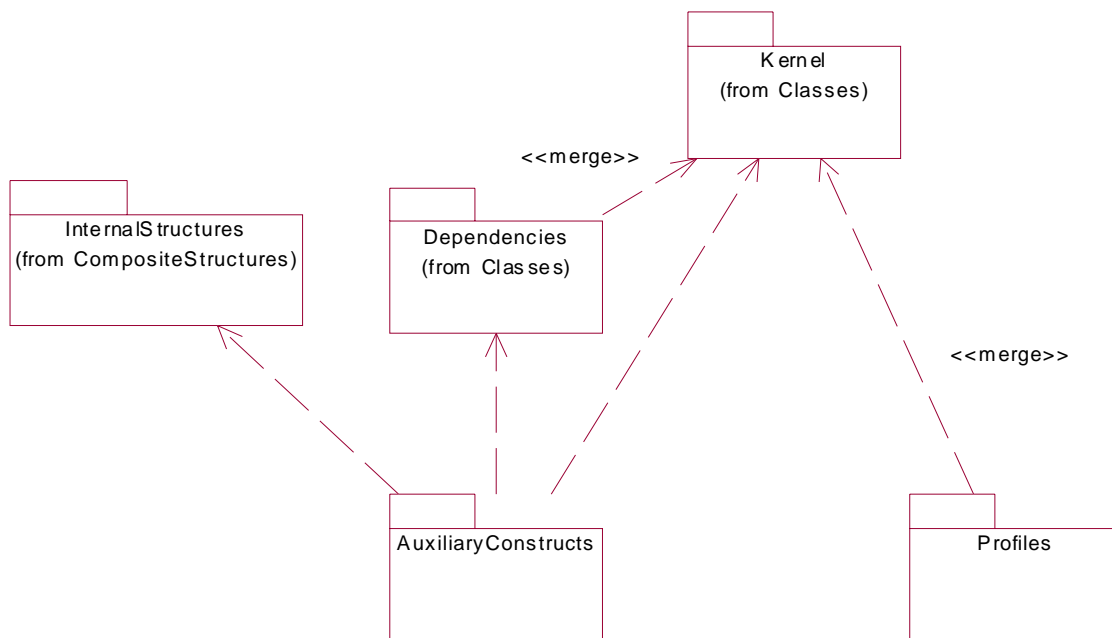
**Figure 16.11 - Use cases owned by a package**

### Changes from previous UML

There are no changes from UML 1.x, although some aspects of notation to model element mapping have been clarified.

## Part III - Supplement

This part defines auxiliary constructs (e.g., information flows, models, templates, primitive types) and the profiles used to customize UML for various domains, platforms, and methods. The UML packages that support auxiliary constructs, along with the structure packages they depend upon (InternalStructures, Dependencies, and Kernel) are shown in the figure below.



**Part III, Figure 1 - UML packages that support auxiliary constructs**

The function and contents of these packages are described in the following chapters, which are organized by major subject areas.





# 17 Auxiliary Constructs

## 17.1 Overview

This chapter defines mechanisms for information flows, models, primitive types, and templates.

### Package structure

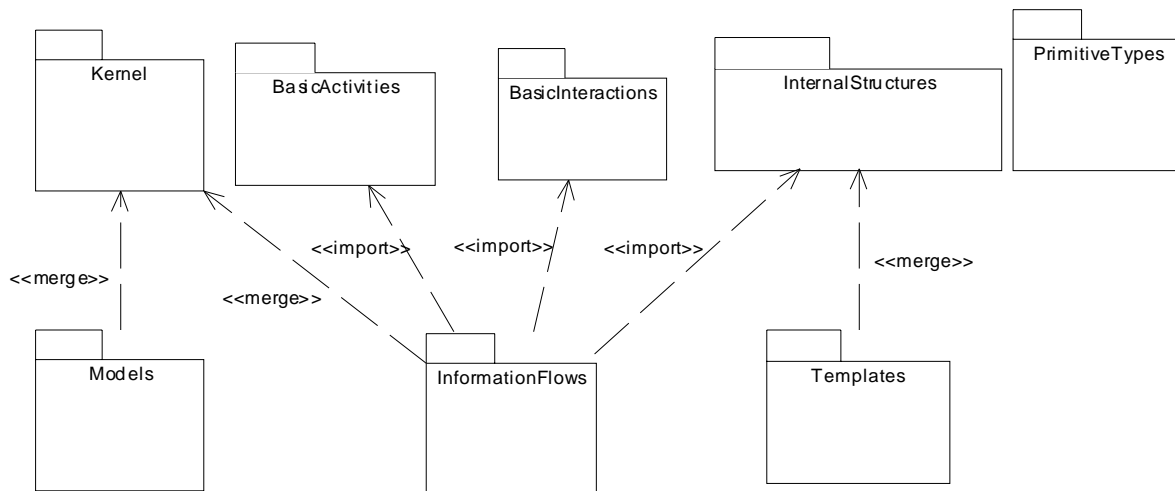


Figure 17.1 - Dependencies between packages described in this chapter

## 17.2 InformationFlows

The InformationFlows package provides mechanisms for specifying the exchange of information between entities of a system at a high level of abstraction. Information flows describe circulation of information in a system in a general manner. They do not specify the nature of the information (type, initial value), nor the mechanisms by which this information is conveyed (message passing, signal, common data store, parameter of operation, etc.). They also do not specify sequences or any control conditions. It is intended that, while modeling in detail, representation and realization links will be able to specify which model element implements the specified information flow, and how the information will be conveyed.

The contents of the InformationFlows package is shown in Figure 17.2. The InformationFlows package is one of the packages of the AuxiliaryConstructs package.

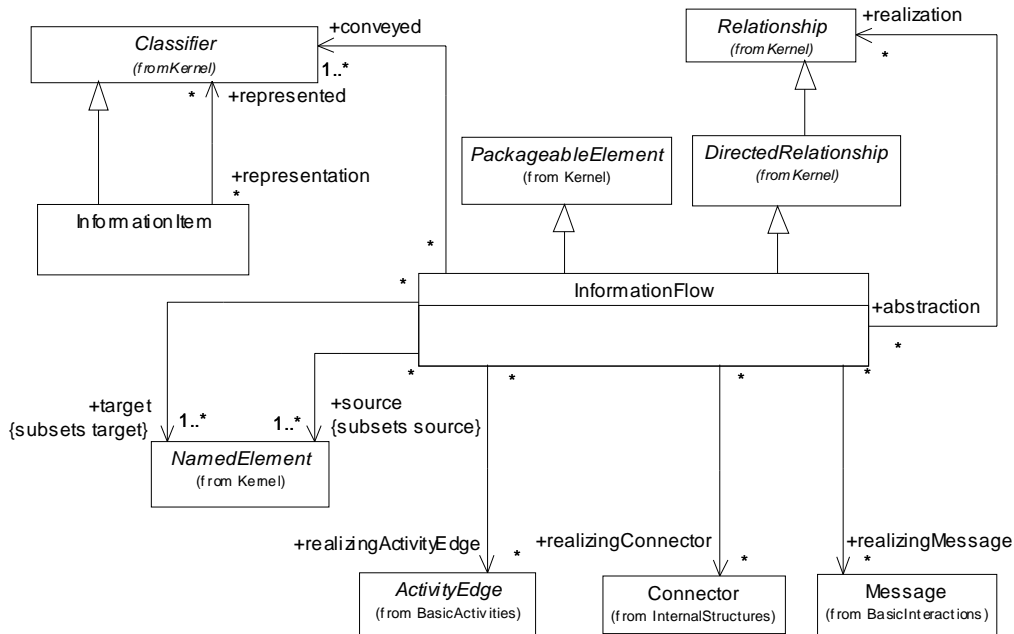


Figure 17.2 - The contents of the InformationFlows package

## 17.2.1 InformationFlow (from InformationFlows)

### Generalizations

- “DirectedRelationship (from Kernel)” on page 59
- “PackageableElement (from Kernel)” on page 105

### Description

An Information Flow specifies that one or more information items circulate from its sources to its targets. Information flows require some kind of “information channel” for transmitting information items from the source to the destination. An information channel is represented in various ways depending on the nature of its sources and targets. It may be represented by connectors, links, associations, or even dependencies. For example, if the source and destination are parts in some composite structure such as a collaboration, then the information channel is likely to be represented by a connector between them. Or, if the source and target are objects (which are a kind of InstanceSpecification), they may be represented by a link that joins the two, and so on.

### Attributes

No additional attributes

### Associations

- realization : Relationship [\*] Determines which Relationship will realize the specified flow.

- conveyed : Classifier [1..\*] Specifies the information items that may circulate on this information flow.
- realizingConnector : Connector[\*] Determines which Connectors will realize the specified flow.
- realizingActivityEdge : ActivityEdge [\*] Determines which ActivityEdges will realize the specified flow.
- realizingMessage : Message[\*] Determines which Messages will realize the specified flow.
- target : NamedElement [1..\*] Defines to which target the conveyed InformationItems are directed.  
{Subsets DirectedRelationship::target}
- source : NamedElement [1..\*] Defines from which source the conveyed InformationItems are initiated.  
{Subsets DirectedRelationship::source}

### Constraints

- [1] The sources and targets of the information flow can only be of the following kind: Actor, Node, UseCase, Artifact, Class, Component, Port, Property, Interface, Package, and InstanceSpecification except when its classifier is a relationship (i.e., it represents a link).
- [2] The sources and targets of the information flow must conform with the sources and targets or conversely the target and sources of the realization relationships, if any.
- [3] An information flow can only convey classifiers that are allowed to represent an information item. (see constraints on InformationItem).

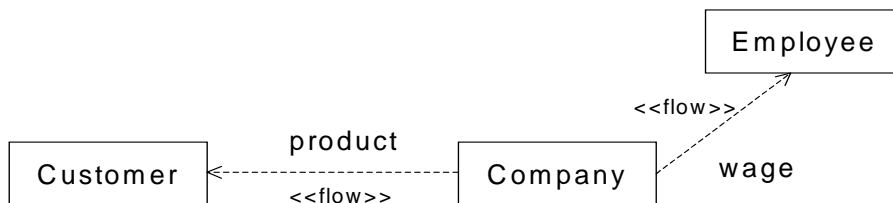
### Semantics

An information flow is an abstraction of the communication of an information item from its sources to its targets. It is used to abstract the communication of information between entities of a system. Sources or targets of an information flow designate sets of objects that can send or receive the conveyed information item. When a source or a target is a classifier, it represents all the potential instances of the classifier; when it is a part, it represents all instances that can play the role specified by the part; when it is a package, it represents all potential instances of the directly or indirectly owned classifiers of the package.

An information flow may directly indicate a concrete classifier, such as a class, that is conveyed instead of using an information item.

### Notation

An information flow is represented as a dependency, with the keyword <<flow>>.



**Figure 17.3 - Example of information flows conveying information items**

### Changes from previous UML

InformationFlow does not exist in UML 1.4.

## 17.2.2 InformationItem (from InformationFlows)

### Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 48

### Description

An information Item is an abstraction of all kinds of information that can be exchanged between objects. It is a kind of classifier intended for representing information at a very abstract way, which cannot be instantiated.

One purpose of Information Items is to be able to define preliminary models, before having taken detailed modeling decisions on types or structures. One other purpose of information items and information flows is to abstract complex models by a less precise but more general representation of the information exchanged between entities of a system.

### Attributes

No additional attributes

### Associations

- represented : Classifier [\*]      Determines the classifiers that will specify the structure and nature of the information.  
An information item represents all its represented classifiers.

### Constraints

- [1] The sources and targets of an information item (its related information flows) must designate subsets of the sources and targets of the representation information item, if any. The Classifiers that can realize an information item can only be of the following kind: Class, Interface, InformationItem, Signal, Component.
- [2] An informationItem has no feature, no generalization, and no associations.
- [3] It is not instantiable.  
isAbstract

### Semantics

“Information” as represented by an information item encompasses all sorts of data, events, facts that are exploited inside the modeled system. For example, the information item “wage” can represent a Salary Class, or a Bonus Class (see example Figure 17.5). An information item does not specify the structure, the type, or the nature of the represented information. It specifies at an abstract level the elements of information that will be exchanged inside a system. More accurate description will be provided by defining the classifiers represented by information item.

Information items can be decomposed into more specific information items, using representation links between them. This gives the ability to express that in specific contexts (specific information flows) a specific information is exchanged.

Information Items cannot have properties, or associations. Specifying these detailed informations belongs to the represented classifiers.

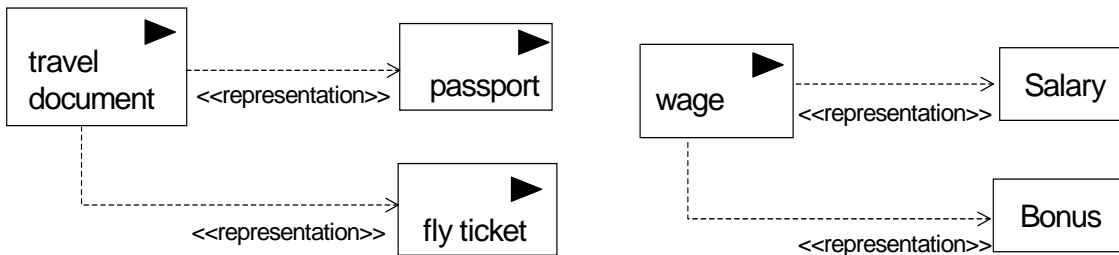
### Notation

Being a classifier, an information item can be represented as a name inside a rectangle. The keyword «information» or the black triangle icon on top of this rectangle indicates that it is an information item.



**Figure 17.4 - Information Item represented as a classifier**

Representation links between a classifier and a representation item are represented as dashed lines with the keyword «representation».



**Figure 17.5 - Examples of «representation» notation**

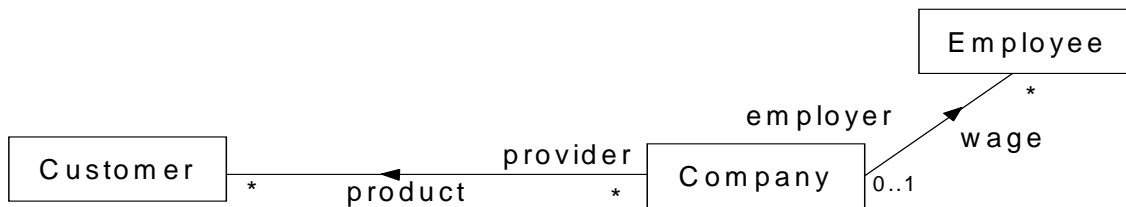
An information item is usually represented attached to an information flow, or to a relationship realizing an information flow. When it is attached to an information flow (see Figure 17.3) its name is displayed close to the information flow line. When it is attached to an information channel, a black triangle on the information channel indicates the direction (source and target) of the realized information flow conveying the information item, and its name is displayed close to that triangle. In the example Figure 17.7, two associations are realizing information flows. The black triangle indicates that an information flow is realized, and the information item name is displayed close to the triangle.

The example Figure 17.6 shows the case of information items represented on connectors. When several information items having the same direction are represented, only one triangle is shown, and the list of information item names, separated by a comma is presented.

The name of the information item can be prefixed by the names of the container elements, such as a container information flow, or a container package or classifier, separated by a colon.



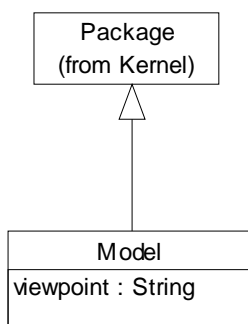
**Figure 17.6 - Information item attached to connectors**



**Figure 17.7 - Information Items attached to associations**

## 17.3 Models

The contents of the Models package is shown in Figure 17.8. The Models package is one of the packages of the AuxiliaryConstructs package.



**Figure 17.8 - The contents of the Models package**

### 17.3.1 Model (from Models)

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

#### Generalizations

- “Package (from Kernel)” on page 103

#### Description

The Model construct is defined as a Package. It contains a (hierarchical) set of elements that together describe the physical system being modeled. A Model may also contain a set of elements that represents the environment of the system, typically Actors, together with their interrelationships, such as Associations and Dependencies.

#### Attributes

- viewpoint : String [\*]      The name of the viewpoint that is expressed by a model. (This name may refer to a profile definition.)

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

A model is a description of a physical system with a certain purpose, such as to describe logical or behavioral aspects of the physical system to a certain category of readers.

Thus, a model is an abstraction of a physical system. It specifies the physical system from a certain vantage point (or viewpoint) for a certain category of stakeholders (e.g., designers, users, or customers of the system) and at a certain level of abstraction, both given by the purpose of the model. A model is complete in the sense that it covers the whole physical system, although only those aspects relevant to its purpose (i.e., within the given level of abstraction and vantage point) are represented in the model. Furthermore, it describes the physical system only once (i.e., there is no overlapping; no part of the physical system is captured more than once in a model).

A model owns or imports all the elements needed to represent a physical system completely according to the purpose of this particular model. The elements are organized into a containment hierarchy where the top-most package or subsystem represents the boundary of the physical system. It is possible to have more than one containment hierarchy within a model (i.e., the model contains a set of top-most packages/subsystems each being the root of a containment hierarchy). In this case there is no single package/subsystem that represents the physical system boundary.

The model may also contain elements describing relevant parts of the system's environment. The environment is typically modeled by actors and their interfaces. As these are external to the physical system, they reside outside the package/subsystem hierarchy. They may be collected in a separate package, or owned directly by the model. These elements and the elements representing the physical system may be associated with each other.

Different models can be defined for the same physical system, where each model represents a view of the physical system defined by its purpose and abstraction level. Typically different models are complementary and defined from the perspectives (viewpoints) of different system stakeholders. When models are nested, the container model represents the comprehensive view of the physical system given by the different views defined by the contained models.

Models can have refinement or mapping dependencies between them. These are typically decomposed into dependencies between the elements contained in the models. Relationships between elements in different models have no semantic impact on the contents of the models because of the self-containment of models. However, they are useful for tracing refinements and for keeping track of requirements between models.

## Notation

A model is notated using the ordinary package symbol (a folder icon) with a small triangle in the upper right corner of the large rectangle. Optionally, especially if contents of the model are shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.

## Presentation Options

A model is notated as a package, using the ordinary package symbol with the keyword «model» placed above the name of the model.



## Examples

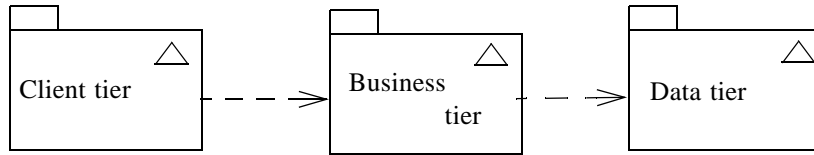


Figure 17.9 - Three models representing parts of a system

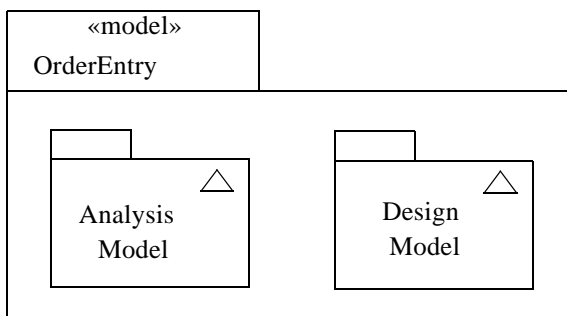


Figure 17.10 - Two views of one and the same physical system collected in a container model

## 17.4 PrimitiveTypes

A number of primitive types have been defined for use in the specification of the UML metamodel. These include primitive types such as Integer, Boolean, and String. These types are reused by both MOF and UML, and may potentially be reused also in user models. Tool vendors, however, typically provide their own libraries of data types to be used when modeling with UML.

The contents of the PrimitiveTypes package is shown in Figure 17.11. The PrimitiveTypes package is one of the packages of the AuxiliaryConstructs package.

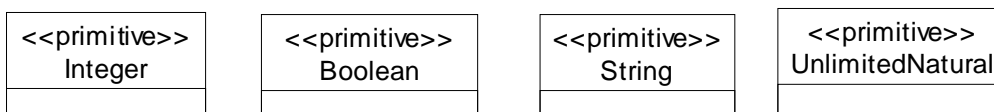


Figure 17.11 - The contents of the PrimitiveTypes package

### 17.4.1 Boolean (from PrimitiveTypes)

A boolean type is used for logical expression, consisting of the predefined values true and false.

#### Generalizations

None

## Description

Boolean is an instance of `PrimitiveType`. In the metamodel, Boolean defines an enumeration that denotes a logical condition. Its enumeration literals are:

- `true` - The Boolean condition is satisfied.
- `false` - The Boolean condition is not satisfied.

It is used for boolean attribute and boolean expressions in the metamodel, such as OCL expression.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

Boolean is an instance of `PrimitiveType`.

## Notation

Boolean will appear as the type of attributes in the metamodel. Boolean instances will be values associated to slots, and can have literally the following values: `true` or `false`.

## Examples

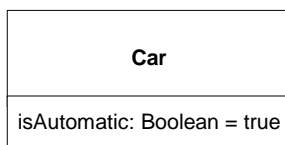


Figure 17.12 - An example of a boolean attribute

## 17.4.2 Integer (from `PrimitiveTypes`)

An integer is a primitive type representing integer values.

## Generalizations

None

### Description

An instance of Integer is an element in the (infinite) set of integers (...-2, -1, 0, 1, 2...). It is used for integer attributes and integer expressions in the metamodel.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

Integer is an instance of PrimitiveType.

### Notation

Integer will appear as the type of attributes in the metamodel. Integer instances will be values associated to slots such as 1, -5, 2, 34, 26524, etc.

### Examples

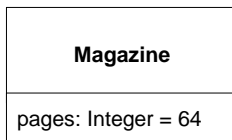


Figure 17.13 - An example of an integer attribute

## 17.4.3 String (from PrimitiveTypes)

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters.

### Generalizations

None

### Description

An instance of String defines a piece of text. The semantics of the string itself depends on its purpose, it can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

String is an instance of PrimitiveType.

### Notation

String appears as the type of attributes in the metamodel. String instances are values associated to slots. The value is a sequence of characters surrounded by double quotes (""). It is assumed that the underlying character set is sufficient for representing multibyte characters in various human languages; in particular, the traditional 8-bit ASCII character set is insufficient. It is assumed that tools and computers manipulate and store strings correctly, including escape conventions for special characters, and this document will assume that arbitrary strings can be used.

A string is displayed as a text string graphic. Normal printable characters should be displayed directly. The display of nonprintable characters is unspecified and platform-dependent.

### Examples

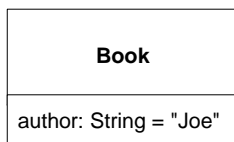


Figure 17.14 - An example of a string attribute

## 17.4.4 UnlimitedNatural (from PrimitiveTypes)

An unlimited natural is a primitive type representing unlimited natural values.

### Generalizations

None

### Description

An instance of UnlimitedNatural is an element in the (infinite) set of naturals (0, 1, 2...). The value of infinity is shown using an asterisk (\*).

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

UnlimitedNatural is an instance of PrimitiveType.

### Notation

UnlimitedNatural will appear as the type of upper bounds of multiplicities in the metamodel. UnlimitedNatural instances will be values associated to slots such as 1, 5, 398475, etc. The value infinity may be shown using an asterisk (\*).

### Examples

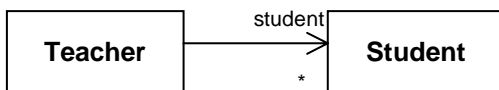


Figure 17.15 - An example of an unlimited natural

## 17.5 Templates

The Templates package specifies how both Classifiers, Packages, and Operations can be parameterized with Classifier, ValueSpecification, and Feature (Property and Operation) template parameters. The package introduces mechanisms for defining templates, template parameters, and bound elements in general, and the specialization of these for classifiers and packages.

Classifier, package, and operation templates were covered in 1.x in the sense that any model element could be templateable. This new metamodel restricts the templateable elements to those for which it is meaningful to have template parameters.

## Templates

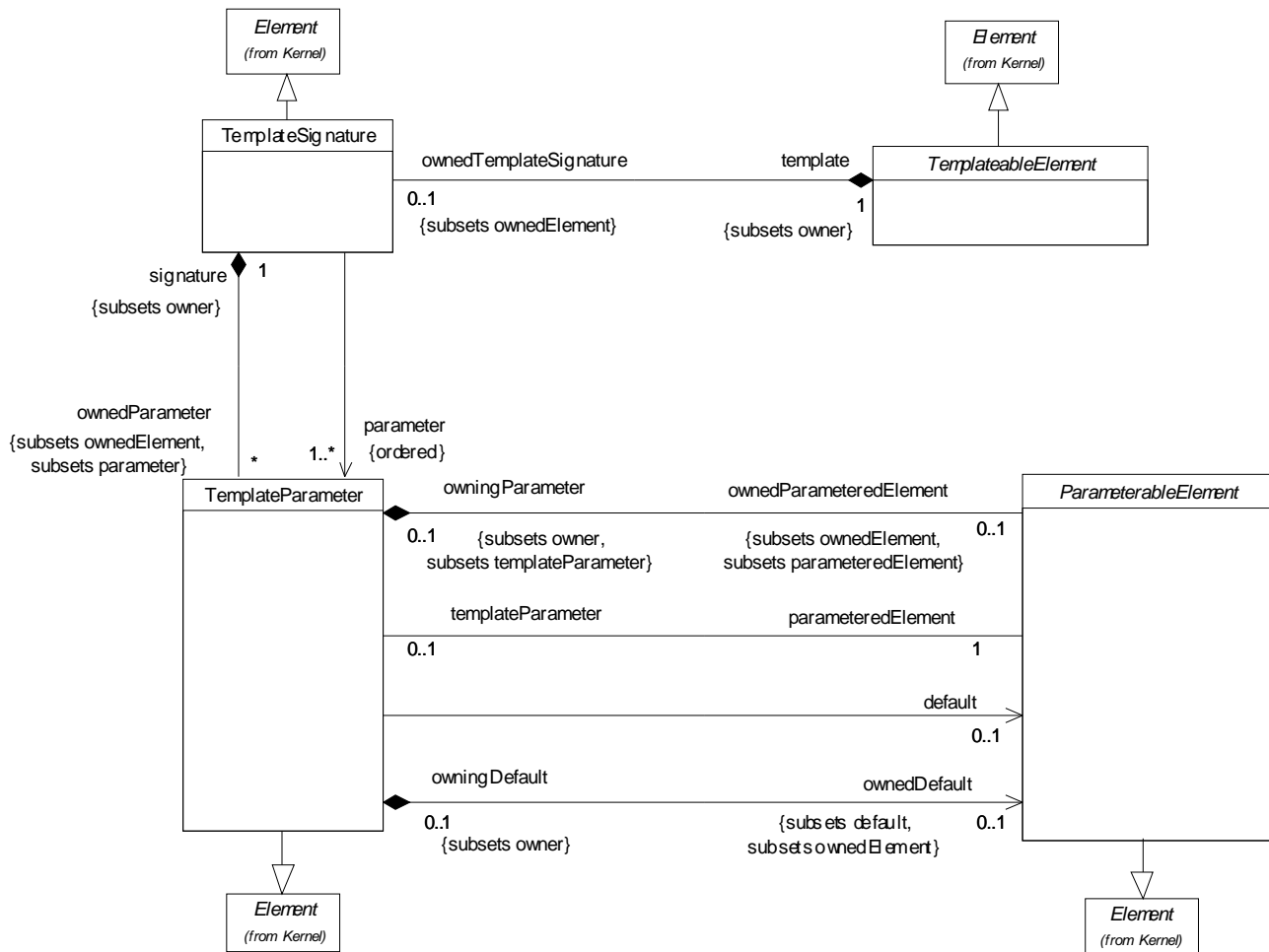


Figure 17.16 - Templates

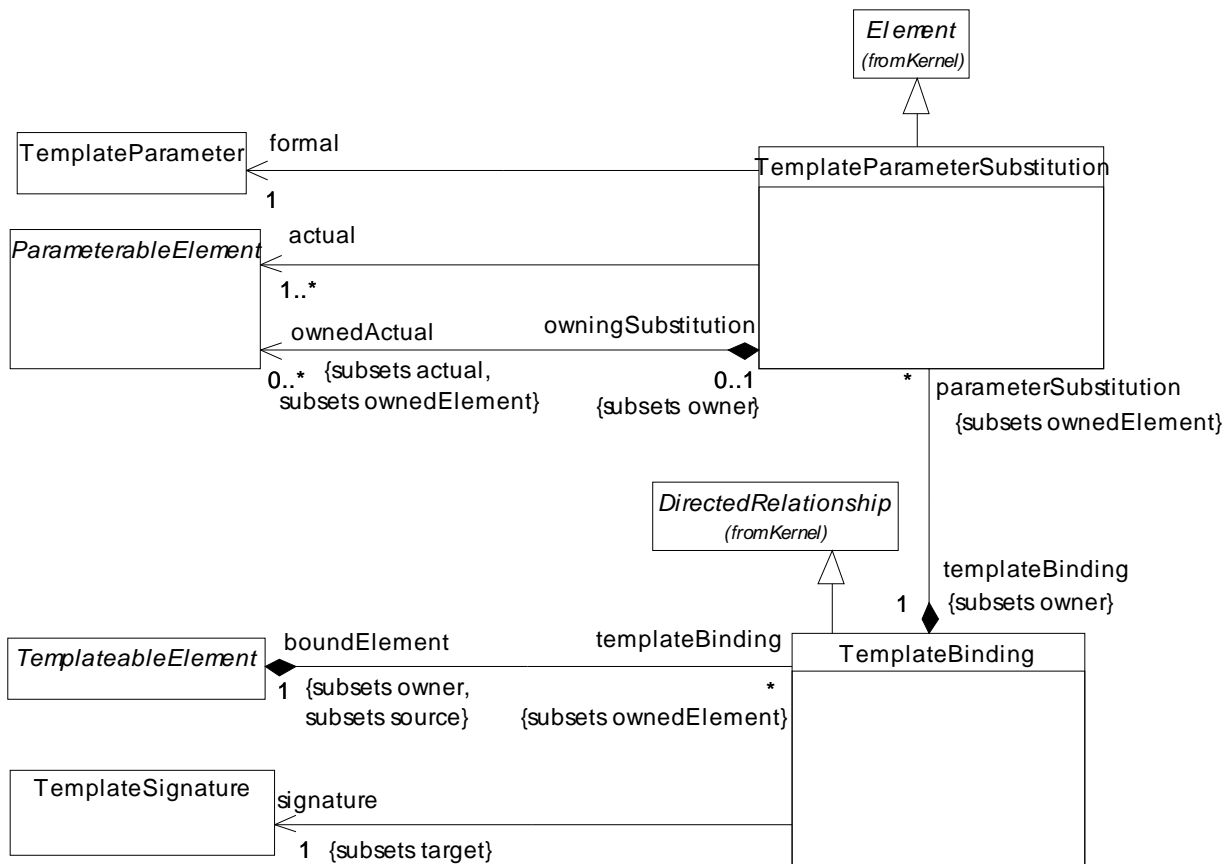


Figure 17.17 - Template parameters

### 17.5.1 ParameterableElement (from Templates)

A parameterable element is an element that can be exposed as a formal template parameter for a template, or specified as an actual parameter in a binding of a template.

#### Generalizations

- “Element (from Kernel)” on page 60

#### Description

A ParameterableElement can be referenced by a TemplateParameter when defining a formal template parameter for a template. A ParameterableElement can be referenced by a TemplateParameterSubstitution when used as an actual parameter in a binding of a template.

ParameterableElement is an abstract metaclass.

### Attributes

No additional attributes

### Associations

- `owningParameter : TemplateParameter[0..1]`  
The formal template parameter that owns this element. Subsets `Element::owner` and `ParameterableElement::templateParameter`.
- `templateParameter : TemplateParameter [0..1]`  
The template parameter that exposes this element as a formal parameter.

### Constraints

No additional constraints

### Additional Operations

- [1] The query `isCompatibleWith()` determines if this parameterable element is compatible with the specified parameterable element. By default parameterable element P is compatible with parameterable element Q if the kind of P is the same or a subtype as the kind of Q. Subclasses should override this operation to specify different compatibility constraints.  
`ParameterableElement::isCompatibleWith(p : ParameterableElement) : Boolean;`  
`isCompatibleWith = p->oclIsKindOf(self.oclType)`
- [2] The query `isTemplateParameter()` determines if this parameterable element is exposed as a formal template parameter.  
`ParameterableElement::isTemplateParameter() : Boolean;`  
`isTemplateParameter = templateParameter->notEmpty()`

### Semantics

A `ParameterableElement` may be part of the definition of a template parameter. The `ParameterableElement` is used to constrain the actual arguments that may be specified for this template parameter in a binding of the template.

A `ParameterableElement` exposed as a template parameter can be used in the template as any other element of this kind defined in the namespace of the template. For example, a classifier template parameter can be used as the type of typed elements. In an element bound to the template, any use of the template parameter will be substituted by the use of the actual parameter.

If a `ParameterableElement` is exposed as a template parameter, then the parameterable element is only meaningful within the template (it may not be used in other parts of the model).

### Semantic Variation Points

The enforcement of template parameter constraints is a semantic variation point:

- If template parameter constraints apply, then within the template element a template parameter can only be used according to its constraint. For example, an operation template parameter can only be called with actual parameters matching the constraint in terms of the signature constraint of the operation template parameter. Applying constraints will imply that a bound element is well-formed if the template element is well-formed and if actual parameters comply with the formal parameter constraints.



- If template parameter constraints do not apply, then within the template element a template parameter can be used without being constrained. For example, an operation template parameter will have no signature in terms of parameters and it can be called with arbitrary actual parameters. Not applying constraints provides more flexibility, but some actual template parameters may not yield a well-formed bound element.

## Notation

See `TemplateParameter` for a description of the notation for exposing a `ParameterableElement` as a formal parameter of a template.

See `TemplateBinding` for a description of the notation for using a `ParameterableElement` as an actual parameter in a binding of a template.

Within these notations, the parameterable element is typically shown as the name of the parametered element (if that element is a named element).

## Examples

See `TemplateParameter`.

## 17.5.2 TemplateableElement (from Templates)

A templateable element is an element that can optionally be defined as a template and bound to other templates.

### Generalizations

- “Element (from Kernel)” on page 60

### Description

`TemplateableElement` may contain a template signature that specifies the formal template parameters. A `TemplateableElement` that contains a template signature is often referred to as a template.

`TemplateableElement` may contain bindings to templates that describe how the templateable element is constructed by replacing the formal template parameters with actual parameters. A `TemplateableElement` containing bindings is often referred to as a bound element.

### Attributes

No additional attributes

### Associations

- `ownedTemplateSignature` : `TemplateSignature`[0..1]  
The optional template signature specifying the formal template parameters. Subsets `Element::ownedElement`.
- `templateBinding` : `TemplateBinding`[\*]  
The optional bindings from this element to templates.

### Constraints

No additional constraints

## Additional Operations

- [1] The query `parameterableElements()` returns the set of elements that may be used as the parametered elements for a template parameter of this templateable element. By default, this set includes all the owned elements. Subclasses may override this operation if they choose to restrict the set of parameterable elements.

```
TemplateableElement::parameterableElements() : Set(ParameterableElement);  
parameterableElements = allOwnedElements->select(oclsIsKindOf(ParameterableElement))
```

- [2] The query `isTemplate()` returns whether this templateable element is actually a template.

```
TemplateableElement::isTemplate() : Boolean;  
isTemplate = ownedSignature->notEmpty()
```

## Semantics

A `TemplateableElement` that has a template signature is a specification of a template. A template is a parameterized element that can be used to generate other model elements using `TemplateBinding` relationships. The template parameters for the template signature specify the formal parameters that will be substituted by actual parameters (or the default) in a binding.

A template parameter is defined in the namespace of the template, but the template parameter represents a model element that is defined in the context of the binding.

A templateable element can be bound to other templates. This is represented by the bound element having bindings to the template signatures of the target templates. In a canonical model a bound element does not explicitly contain the model elements implied by expanding the templates it binds to, since those expansions are regarded as derived. The semantics and well-formedness rules for the bound element must be evaluated as if the bindings were expanded with the substitutions of actual elements for formal parameters.

The semantics of a binding relationship is equivalent to the model elements that would result from copying the contents of the template into the bound element, replacing any elements exposed as a template parameter with the corresponding element(s) specified as actual parameters in this binding.

A bound element may have multiple bindings, possibly to the same template. In addition, the bound element may contain elements other than the bindings. The specific details of how the expansions of multiple bindings, and any other elements owned by the bound element, are combined together to fully specify the bound element are found in the subclasses of `TemplateableElement`. The general principle is that one evaluates the bindings in isolation to produce intermediate results (one for each binding), which are then merged to produce the final result. It is the way the merging is done that is specific to each kind of templateable element.

A templateable element may contain both a template signature and bindings. Thus a templateable element may be both a template and a bound element.

A template cannot be used in the same manner as a non-template element of the same kind. The template element can only be used to generate bound elements (e.g., a template class cannot be used as the type of a typed element) or as part of the specification of another template (e.g., a template class may specialize another template class).

A bound (non-template) element is an ordinary element and can be used in the same manner as a non-bound (and non-template) element of the same kind. For example, a bound class may be used as the type of a typed element.

## Notation

If a `TemplateableElement` has template parameters, a small dashed rectangle is superimposed on the symbol for the templateable element, typically on the upper right-hand corner of the notation (if possible). The dashed rectangle contains a list of the formal template parameters. The parameter list must not be empty, although it might be suppressed in the presentation. Any other compartments in the notation of the templateable element will appear as normal.

The formal template parameter list may be shown as a comma-separated list, or it may be one formal template parameter per line. See `TemplateParameter` for the general syntax of each template parameter.

A bound element has the same graphical notation as other elements of that kind. Each binding is shown using the notation described under `TemplateBinding`.

## Presentation Options

An alternative presentation for the bindings for a bound element is to include the binding information within the notation for the bound element. Typically the name compartment would be extended to contain a string with the following syntax:

```
<element-name> ‘.’ <binding-expression> [‘,’ <binding-expression>]*  
<binding-expression> ::= <template-element-name> ‘<’ <template-parameter-substitution>  
[‘,’ <template-parameter-substitution>]* ‘>’
```

and `<template-parameter-substitution>` is defined in `TemplateBinding` (from `Templates`).

## Examples

For examples of templates, the reader is referred to those sections that deal with specializations of `TemplateableElement`, in particular `ClassifierTemplate` and `PackageTemplate`.

### 17.5.3 TemplateBinding (from Templates)

A template binding represents a relationship between a templateable element and a template. A template binding specifies the substitutions of actual parameters for the formal parameters of the template.

## Generalizations

- “`DirectedRelationship` (from `Kernel`)” on page 59

## Description

`TemplateBinding` is a directed relationship from a bound templateable element to the template signature of the target template. A `TemplateBinding` owns a set of template parameter substitutions.

## Attributes

No additional attributes

## Associations

- `parameterSubstitution` : `TemplateParameterSubstitution`[\*]  
The parameter substitutions owned by this template binding. Subsets `Element::ownedElement`.
- `boundElement` : `TemplateableElement`[1]  
The element that is bound by this binding. Subsets `DirectedRelationship::source`.

- `template : TemplateSignature[1]`  
The template signature for the template that is the target of the binding. Subsets `DirectedRelationship::target`.

## Constraints

- [1] Each parameter substitution must refer to a formal template parameter of the target template signature.  
`parameterSubstitution->forall(b | template.parameter->includes(b.formal))`
- [2] A binding contains at most one parameter substitution for each formal template parameter of the target template signature.  
`template.parameter->forall(p | parameterSubstitution->select(b | b.formal = p)->size() <= 1)`

## Semantics

The presence of a `TemplateBinding` relationship implies the same semantics as if the contents of the template owning the target template signature were copied into the bound element, substituting any elements exposed as formal template parameters by the corresponding elements specified as actual parameters in this binding. If no actual parameter is specified in this binding for a formal parameter, then the default element for that formal template parameter (if specified) is used.

## Semantic Variation Points

It is a semantic variation point

- if all formal template parameters must be bound as part of a binding (complete binding), or
- if a subset of the formal template parameters may be bound in a binding (partial binding).

In case of complete binding, the bound element may have its own formal template parameters, and these template parameters can be provided as actual parameters of the binding. In case of partial binding, the unbound formal template parameters are formal template parameters of the bound element.

## Notation

A `TemplateBinding` is shown as a dashed arrow with the tail on the bound element and the arrowhead on the template and the keyword «bind». The binding information is generally displayed as a comma-separated list of template parameter substitutions:

`<template-param-substitution> ::= <template-param-name> ‘->’ <actual-template-parameter>`

where the syntax of `<template-param-name>` depends on the kind of `ParameteredElement` for this template parameter substitution and the kind of `<actual-template-parameter>` depends upon the kind of element. See “`ParameterableElement` (from `Templates`)” on page 602 (and its subclasses).

## Examples

For examples of templates, the reader is referred to those sections that deal with specializations of `TemplateableElement`, in particular `ClassifierTemplate` and `PackageTemplate`.

### 17.5.4 TemplateParameter (from Templates)

A template parameter exposes a parameterable element as a formal template parameter of a template.

## Generalizations

- “Element (from Kernel)” on page 60

## Description

TemplateParameter references a ParameterableElement that is exposed as a formal template parameter in the containing template.

## Attributes

No additional attributes

## Associations

- default : ParameterableElement[0..1]  
The element that is the default for this formal template parameter.
- ownedDefault : ParameterableElement[0..1]  
The element that is owned by this template parameter for the purpose of providing a default. Subsets default and Element::ownedElement.
- ownedParameteredElement : ParameterableElement[0..1]  
The element that is owned by this template parameter. Subsets parameteredElement and Element::ownedElement.
- parameteredElement : ParameterableElement[1]  
The element exposed by this template parameter.
- signature : TemplateSignature[1]  
The template signature that owns this template parameter. Subsets Element::owner.

## Constraints

- [1] The default must be compatible with the formal template parameter.  
default->notEmpty() implies default->isCompatibleWith(parameteredElement)
- [2] The default can only be owned if the parametered element is not owned.  
ownedDefault->notEmpty() implies ownedParameteredElement->isEmpty()

## Semantics

A TemplateParameter references a ParameterableElement that is exposed as a formal template parameter in the containing template. This parameterable element is meaningful only within the template, or other templates that may have access to its internals (e.g., if the template supports specialization). The exposed parameterable element may not be used in other parts of the model. A TemplateParameter may own the exposed ParameterableElement in situations where that element is only referenced from within the template.

Each exposed element constrains the elements that may be substituted as actual parameters in a binding.

A TemplateParameter may reference a ParameterableElement as the default for this formal parameter in any binding that does not provide an explicit substitution. The TemplateParameter may own this default ParameterableElement in situations where the exposed ParameterableElement is not owned by the TemplateParameter.

## Notation

The general notation for a template parameter is a string displayed within the template parameter list for the template: