

Generalizations

- “ControlNode (from BasicActivities)” on page 346

Description

A join node has multiple incoming edges and one outgoing edge.

(CompleteActivities) Join nodes have a boolean value specification using the names of the incoming edges to specify the conditions under which the join will emit a token.

Attributes

Package CompleteActivities

- `isCombineDuplicate` : Boolean [1..1] Tells whether tokens having objects with the same identity are combined into one by the join. Default value is true.

Associations

Package CompleteActivities

- `joinSpec` : ValueSpecification [1..1] A specification giving the conditions under which the join will emit a token. Default is “and.”

Constraints

[1] A join node has one outgoing edge.

[2] If a join node has an incoming object flow, it must have an outgoing object flow, otherwise, it must have an outgoing control flow.

Semantics

If there is a token offered on all incoming edges, then tokens are offered on the outgoing edge according to the following join rules:

1. If all the tokens offered on the incoming edges are control tokens, then one control token is offered on the outgoing edge.
2. If some of the tokens offered on the incoming edges are control tokens and others are data tokens, then only the data tokens are offered on the outgoing edge. Tokens are offered on the outgoing edge in the same order they were offered to the join.

Multiple control tokens offered on the same incoming edge are combined into one before applying the above rules. No joining of tokens is necessary if there is only one incoming edge, but it is not a useful case.

Package CompleteActivities

The reserved string “and” used as a join specification is equivalent to a specification that requires at least one token offered on each incoming edge. It is the default. The join specification is evaluated whenever a new token is offered on any incoming edge. The evaluation is not interrupted by any new tokens offered during the evaluation, nor are concurrent evaluations started when new tokens are offered during an evaluation.

If any tokens are offered to the outgoing edge, they must be accepted or rejected for traversal before any more tokens are offered to the outgoing edge. If tokens are rejected for traversal, they are no longer offered to the outgoing edge. The join specification may contain the names of the incoming edges to refer to whether a token was offered on that edge at the time the evaluation started.

If `isCombinedDuplicate` is true, then before object tokens are offered to the outgoing edge, those containing objects with the same identity are combined into one token.

Other rules for when tokens may be passed along the outgoing edge depend on the characteristics of the edge and its target. For example, if the outgoing edge targets an object node that has reached its upper bound, no token can be passed. The rules may be optimized to a different algorithm as long as the effect is the same. In the full object node example, the implementation can omit the unnecessary join evaluations until the down stream object node can accept tokens.

Notation

The notation for a join node is a line segment, as illustrated on the left side of the figure below. The join node must have one or more activity edges entering it, and only one edge leaving it. The functionality of join node and fork node can be combined by using the same node symbol, as illustrated at the right side of the figure below. This case maps to a model containing a join node with all the incoming edges shown in the diagram and one outgoing edge to a fork node that has all the outgoing edges shown in the diagram. It assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

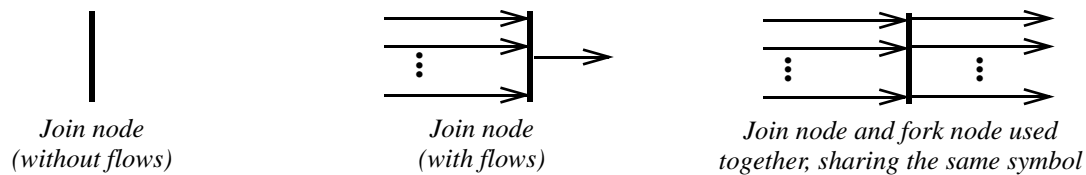


Figure 12.101 - Join node notations

Package CompleteActivities

Join specifications are shown near the join node, as shown below.

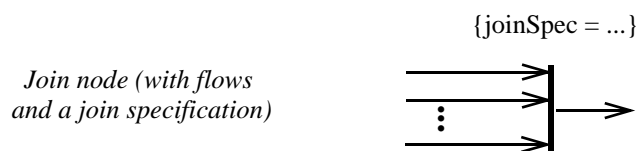


Figure 12.102 - Join node notations

Examples

The example at the left of the figure indicates that a Join is used to synchronize the processing of the Ship Order and Accept Order behaviors. Here, when both have been completed, control is passed to Close Order.

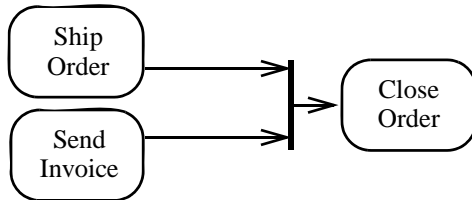


Figure 12.103 - Join node example

Package CompleteActivities

The example below illustrates how a join specification can be used to ensure that both a drink is selected and the correct amount of money has been inserted before the drink is dispensed. Names of the incoming edges are used in the join specification to refer to whether tokens are available on the edges.

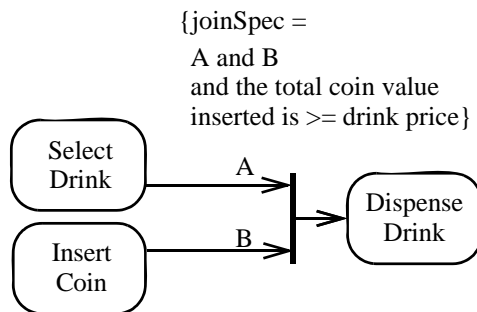


Figure 12.104 - Join node example

Rationale

Join nodes are introduced to support parallelism in activities.

Changes from previous UML

Join nodes replace the use of PseudoState with join kind in UML 1.5 activity modeling.

12.3.35 LoopNode (from CompleteStructuredActivities, StructuredActivities)

A loop node is a structured activity node that represents a loop with setup, test, and body sections.

Generalizations

- “StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)” on page 396.

Description

Each section is a well-nested subregion of the activity whose nodes follow any predecessors of the loop and precede any successors of the loop. The test section may precede or follow the body section. The setup section is executed once on entry to the loop, and the test and body sections are executed repeatedly until the test produces a false value. The results of the final execution of the test or body are available after completion of execution of the loop.

Attributes

- `isTestedFirst` : Boolean [1] If true, the test is performed before the first execution of the body. If false, the body is executed once before the test is performed.

Associations

Package StructuredActivities

- `setupPart` : ActivityNode[0..*] The set of nodes and edges that initialize values or perform other setup computations for the loop.
- `bodyPart` : ActivityNode[0..*] The set of nodes and edges that perform the repetitive computations of the loop. The body section is executed as long as the test section produces a true value.
- `test` : ActivityNode[0..*] The set of nodes, edges, and designated value that compute a Boolean value to determine if another execution of the body will be performed.
- `decider` : OutputPin [1] An output pin within the test fragment the value of which is examined after execution of the test to determine whether to execute the loop body.

Package CompleteStructuredActivities

- `result` : OutputPin [0..*] A list of output pins that constitute the data flow output of the entire loop.
- `loopVariable` : OutputPin [0..*] A list of output pins owned by the loop that hold the values of the loop variables during an execution of the loop. When the test fails, the values are copied to the result pins of the loop.
- `bodyOutput` : OutputPin [0..*] A list of output pins within the body fragment the values of which are copied to the loop variable pins after completion of execution of the body, before the next iteration of the loop begins or before the loop exits.
- `loopVariableInput` : InputPin[0..*] A list of values that are copied into the loop variable pins before the first iteration of the loop.

Constraints

No additional constraints

Semantics

No part of a loop node is executed until all control-flow or data-flow predecessors of the loop node have completed execution. When all such predecessors have completed execution and made tokens available to inputs of the loop node, the loop node captures the input tokens and begins execution.

First the setup section of the loop node is executed. A *front end node* is a node within a nested section (such as the setup section, test section, or body section) that has no predecessor dependencies within the same section. A control token is offered to each front end node within the setup section. Nodes in the setup section may also have individual dependencies (typically data flow dependencies) on nodes external to the loop node. To begin execution, such nodes must receive their individual tokens in addition to the control token from the overall loop.

A *back end node* is a node within a nested section that has no successor dependencies within the same section. When all the back end nodes have completed execution, the overall section is considered to have completed execution. (It may be thought of as delivering a control token to the next section within the loop.)

When the setup section has completed execution, the iterative execution of the loop begins. The test section may precede or follow the body section (test-first loop or test-last loop). The following description assumes that the test section comes first. If the body section comes first, it is always executed at least once, after which this description applies to subsequent iterations.

When the setup section has completed execution (if the test comes first) or when the body section has completed execution of an iteration, the test section is executed. A control token is offered to each front end node within the test section. When all back end nodes in the test section have completed execution, execution of the test section is complete. Typically there will only be one back end node and it will have a Boolean value, but for generality it is permitted to perform arbitrary computation in the test section.

When the test section has completed execution, the Boolean value on the designated *decider* pin within the test section is examined. If the value is true, the body section is executed again. If the value is false, execution of the loop node is complete.

When the setup section has completed execution (if the body comes first) or when the iteration section has completed execution and produced a true value, execution of the body section begins. Each front end node in the body section is offered a control token. When all back end nodes in the body section have completed execution, execution of the body section is complete.

Within the body section, variables defined in the loop node or in some higher-level enclosing node are updated with any new values produced during the iteration and any temporary values are discarded.

Notation

No specific notation.

Rationale

Loop nodes are introduced to provide a structured way to represent iteration.

Changes from previous UML

Loop nodes are new in UML 2.0.

12.3.36 MergeNode (from IntermediateActivities)

A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.

Generalizations

- “ControlNode (from BasicActivities)” on page 346

Description

A merge node has multiple incoming edges and a single outgoing edge.

Attributes

No additional attributes

Associations

No additional associations

Constraints

- [1] A merge node has one outgoing edge.
- [2] The edges coming into and out of a merge node must be either all object flows or all control flows.

Semantics

All tokens offered on incoming edges are offered to the outgoing edge. There is no synchronization of flows or joining of tokens.

Notation

The notation for a merge node is a diamond-shaped symbol, as illustrated on the left side of the figure below. In usage, however, the merge node must have two or more edges entering it and a single activity edge leaving it. The functionality of merge node and decision node can be combined by using the same node symbol, as illustrated at the right side of the figure below. This case maps to a model containing a merge node with all the incoming edges shown in the diagram and one outgoing edge to a decision node that has all the outgoing edges shown in the diagram. It assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

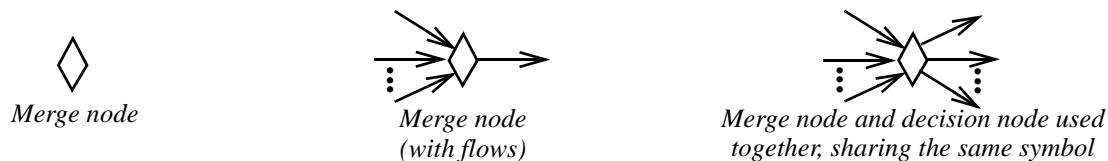


Figure 12.105 - Merge node notation

Examples

In the example below, either one or both of the behaviors, Buy Item or Make Item could have been invoked. As *each* completes, control is passed to Ship Item. That is, if only one of Buy Item or Make Item completes, then Ship Item is invoked only once; if both complete, Ship Item is invoked twice.

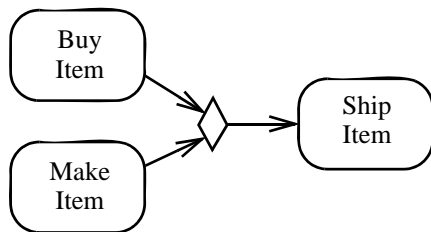


Figure 12.106 - Merge node example

Rationale

Merge nodes are introduced to support bringing multiple flows together in activities. For example, if a decision is used after a fork, the two flows coming out of the decision need to be merged into one before going to a join. Otherwise the join will wait for both flows, only one of which will arrive.

Changes from previous UML

Merge nodes replace the use of PseudoState with junction kind in UML 1.5 activity modeling.

12.3.37 ObjectFlow (from BasicActivities, CompleteActivities)

An object flow is an activity edge that can have objects or data passing along it.

Generalizations

- “ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)” on page 315.

Description

An object flow models the flow of values to or from object nodes.

Package CompleteActivities

Object flows add support for multicast/receive, token selection from object nodes, and transformation of tokens.

Attributes

Package CompleteActivities

- isMulticast : Boolean [1..1] = false Tells whether the objects in the flow are passed by multicasting.
- isMultireceive : Boolean [1..1] = false Tells whether the objects in the flow are gathered from respondents to multicasting.

Associations

Package CompleteActivities

- selection : Behavior [0..1] Selects tokens from a source object node.
- transformation : Behavior [0..1] Changes or replaces data tokens flowing along edge.

Constraints

Package BasicActivities

- [1] Object flows may not have actions at either end.
- [2] Object nodes connected by an object flow, with optionally intervening control nodes, must have compatible types. In particular, the downstream object node type must be the same or a supertype of the upstream object node type.
- [3] Object nodes connected by an object flow, with optionally intervening control nodes, must have the same upper bounds.

Package CompleteActivities

- [1] An edge with constant weight may not target an object node, or lead to an object node downstream with no intervening actions, that has an upper bound less than the weight.
- [2] A transformation behavior has one input parameter and one output parameter. The input parameter must be the same or a supertype of the type of object token coming from the source end. The output parameter must be the same or a subtype of the type of object token expected downstream. The behavior cannot have side effects.
- [3] An object flow may have a selection behavior only if it has an object node as a source.
- [4] A selection behavior has one input parameter and one output parameter. The input parameter must be a bag of elements of the same or a supertype of the type of source object node. The output parameter must be the same or a subtype of the type of source object node. The behavior cannot have side effects.
- [5] isMulticast and isMultireceive cannot both be true.

Semantics

Package BasicActivities

See semantics inherited from ActivityEdge. An object flow is an activity edge that only passes object and data tokens. Tokens offered by the source node are all offered to the target node, subject to the restrictions inherited from ActivityEdge.

Two object flows may have the same object node as source. In this case the edges will compete for objects. Once an edge takes an object from an object node, the other edges do not have access to it. Use a fork to duplicate tokens for multiple uses.

Package CompleteActivities

If a transformation behavior is specified, then each token offered to the edge is passed to the behavior, and the output of the behavior is given to the target node for consideration instead of the token that was input to the transformation behavior. Because the behavior is used while offering tokens to the target node, it may be run many times on the same token before the token is accepted by the target node. This means the behavior cannot have side effects. It may not modify objects, but it may for example, navigate from one object to another, get an attribute value from an object, or replace a data value with another. Transformation behaviors with an output parameter with multiplicity greater than 1 may replace one token with many.

If a selection behavior is specified, then it is used to offer a token from the source object node to the edge, rather than using object node's ordering. It has the same semantics as selection behavior on object nodes. See *ObjectNode*. See application at *DataStoreNode*.

Multicasting and receiving is used in conjunction with partitions to model flows between behaviors that are the responsibility of objects determined by a publish and subscribe facility. To support execution the model must be refined to specify the particular publish/subscribe facility employed. This is illustrated in the Figure 12.113 on page 379.

Notation

An object flow is notated by an arrowed line.

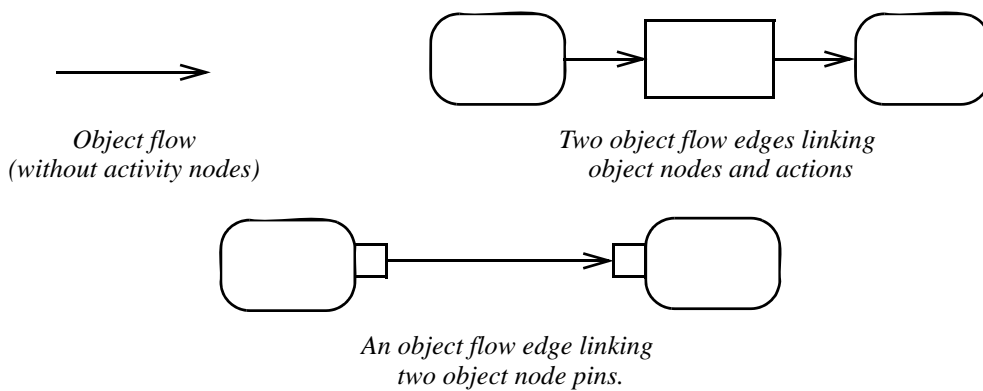


Figure 12.107 - Object flow notations

Package *CompleteActivities*

Selection behavior is specified with the keyword «selection» placed in a note symbol, and attached to the appropriate objectFlow symbol as illustrated in the figure below.

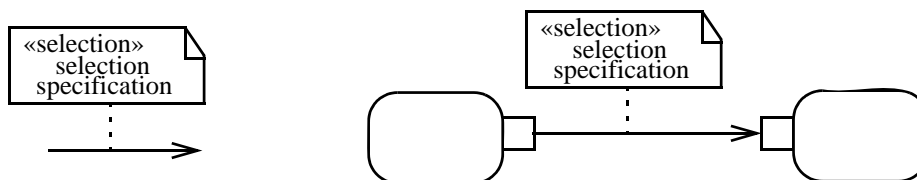


Figure 12.108 - Specifying selection behavior on an Object flow

Presentation Options

To reduce clutter in complex diagrams, object nodes may be elided. The names of the invoked behaviors can suggest their parameters. Tools may support hyperlinking from the edge lines to show the data flowing along them, and show a small square above the line to indicate that pins are elided, as illustrated in the figure below. Any adornments that would normally be near the pin, like effect, can be displayed at the ends of the flow lines.

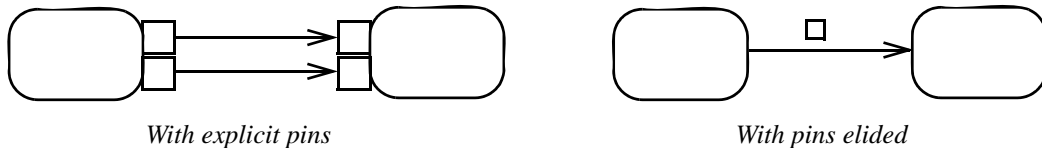


Figure 12.109 - Eliding objects flowing on the edge

Examples

In the example on the left below, the two arrowed lines are both object flow edges. This indicates that order objects flow from Fill Order to Ship Order. In the example on the right, the one arrowed line starts from the Fill Order object node pin and ends at Ship Order object node pin. This also indicates that order objects flow from Fill Order to Ship Order.



Figure 12.110 - Object flow example

On the left, the example below shows the Pick Materials activity provides an order along with its associated materials for assembly. On the right, the object flow has been simplified through eliding the object flow details.

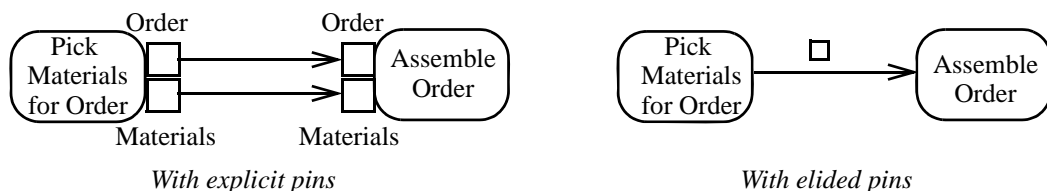


Figure 12.111 - Eliding objects flowing on the edge

Package CompleteActivities

In the figure below, two examples of selection behavior are illustrated. The example on the left indicates that the orders are to be shipped based on order priority—and those with the same priority should be filled on a first-in/first-out (FIFO) basis. The example on the right indicates that the result of a Close Order activity produces closed order objects, but the Send Customer Notice activity requires a customer object. The selection, then, specifies that a query operation that takes an Order evaluates the customer object via the Order.customer:Party association.

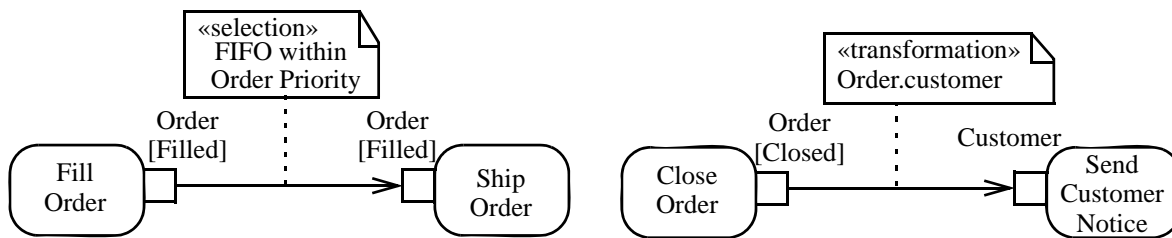


Figure 12.112 - Specifying selection behavior on an Object flow

In the example below, the Requests for Quote (RFQs) are sent to multiple specific sellers (i.e., is multicast) for a quote response by each of the sellers. Some number of sellers then respond by returning their quote response. Since multiple responses can be received, the edge is labeled for the multiple-receipt option. Publish/subscribe and other brokered mechanisms can be handled using the multicast and multireceive mechanisms. Note that the swimlanes are an important feature for indicating the subject and source of this.

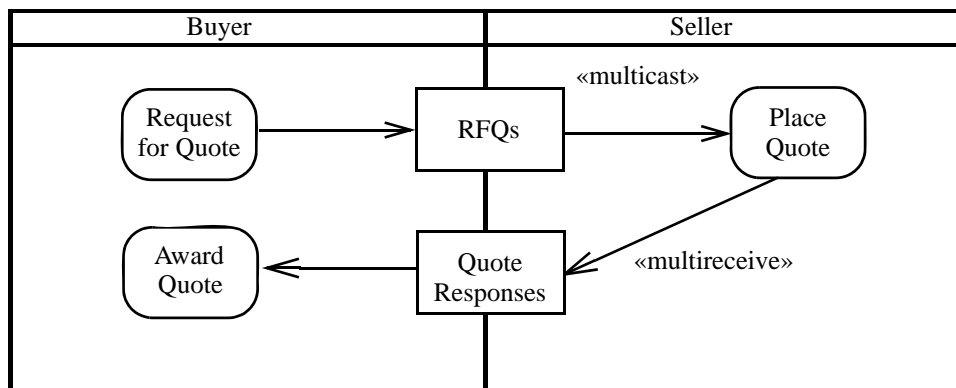


Figure 12.113 - Specifying multicast and multireceive on the edge

Rationale

Object flow is introduced to model the flow of data and objects in an activity.

Changes from previous UML

Explicitly modeled object flows are new in UML 2.0. They replace the use of (state) Transition in UML 1.5 activity modeling. They also replace data flow dependencies from UML 1.5 action model.

12.3.38 ObjectNode (from BasicActivities, CompleteActivities)

An object node is an abstract activity node that is part of defining object flow in an activity.

Generalizations

- “ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, StructuredActivities)” on page 323
- “TypedElement (from Kernel)” on page 131

Description

An object node is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to, as described in the semantics section.

Package CompleteActivities

Complete object nodes add support for token selection, limitation on the number of tokens, specifying the state required for tokens, and carrying control values.

Attributes

Package CompleteActivities

- `ordering : ObjectNodeOrderingKind [1..1] = FIFO` Tells whether and how the tokens in the object node are ordered for selection to traverse edges outgoing from the object node.
- `isControlType : Boolean [1..1] = false` Tells whether the type of the object node is to be treated as control.

Associations

Package CompleteActivities

- `inState : State [0..*]` The required states of the object available at this point in the activity.
- `selection : Behavior [0..1]` Selects tokens for outgoing edges.
- `upperBound : ValueSpecification [1..1] = *` The maximum number of tokens allowed in the node. Objects cannot flow into the node if the upper bound is reached.

Constraints

Package BasicActivities

- [1] All edges coming into or going out of object nodes must be object flow edges.
- [2] Object nodes are not unique typed elements.
`isUnique = false`

Package CompleteActivities

- [1] The upper bound must be equal to the upper bound of nearest upstream and downstream object nodes that do not have intervening action nodes.
- [2] If an object node has a selection behavior, then the ordering of the object node is ordered and vice versa.

- [3] A selection behavior has one input parameter and one output parameter. The input parameter must be a bag of elements of the same type as the object node or a supertype of the type of object node. The output parameter must be the same or a subtype of the type of object node. The behavior cannot have side effects.

Semantics

Object nodes may only contain values at runtime that conform to the type of the object node, in the state or states specified, if any. If no type is specified, then the values may be of any type. Multiple tokens containing the same value may reside in the object node at the same time. This includes data values. A token in an object node can traverse only one of the outgoing edges.

An object node may indicate that its type is to be treated as a control value, even if no type is specified for the node. Control edges may be used with the object node having control type.

Package CompleteActivities

An object node may not contain more tokens than its upper bound. The upper bound must be a `LiteralUnlimitedNatural`. An upper bound of * means the upper bound is unlimited. See `ObjectFlow` for additional rules regarding when objects may traverse the edges incoming and outgoing from an object node.

The ordering of an object node specifies the order in which tokens in the node are offered to the outgoing edges. This can be set to require that tokens do not overtake each other as they pass through the node (FIFO), or that they do (LIFO or modeler-defined ordering). Modeler-defined ordering is indicated by an ordering value of `ordered`, and a selection behavior that determines what token to offer to the edges. The selection behavior takes all the tokens in the object node as input and chooses a single token from those. It is executed whenever a token is to be offered to an edge. Because the behavior is used while offering tokens to outgoing edges, it may be run many times on the same token before the token is accepted by those edges. This means the behavior cannot have side effects. The selection behavior of an object node is overridden by any selection behaviors on its outgoing edges. See `ObjectFlow`. Overtaking due to ordering is distinguished from the case where each invocation of the activity is handled by a separate execution of the activity. In this case, the tokens have no interaction with each other, because they flow through separate executions of the activity. See `Activity`.

Notation

Object nodes are notated as rectangles. A name labeling the node is placed inside the symbol, where the name indicates the type of the object node, or the name and type of the node in the format “name:type.” Object nodes whose instances are sets of the “name” type are labeled as such. Object nodes with a signal as type are shown with the symbol on the right.

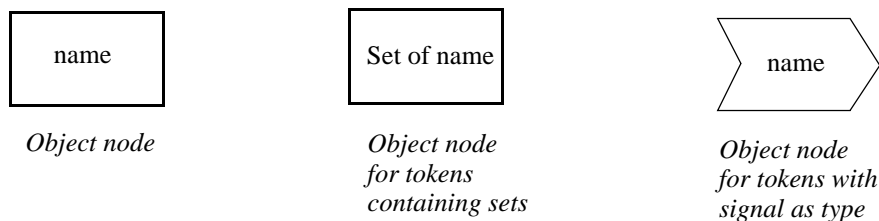


Figure 12.114 - Object node notations

Package CompleteActivities

A name labeling the node indicates the type of the object node. The name can also be qualified by a state or states, which is to be written within brackets below the name of the type. Upper bounds, ordering, and control type other than the defaults are notated in braces underneath the object node.

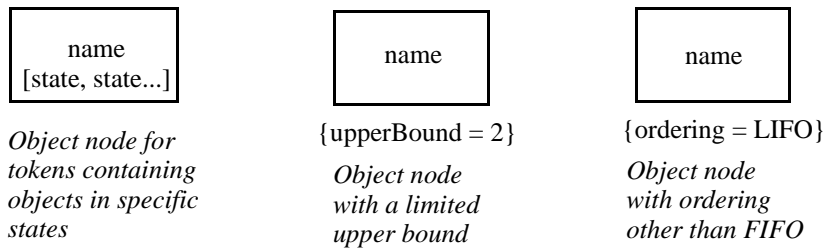


Figure 12.115 - Object node notations

Selection behavior is specified with the keyword «selection» placed in a note symbol, and attached to an ObjectNode symbol as illustrated in the figure below.

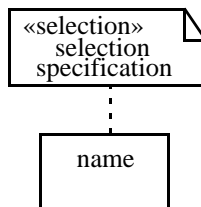


Figure 12.116 - Specifying selection behavior on an Object node

Presentation Options

It is expected that the UML 2.0 Diagram Interchange specification will define a metaassociation between model elements and view elements, like diagrams. It can be used to link an object node to an object diagram showing the classifier that is the type of the object and its relations to other elements. Tools can use this information in various ways to integrate the activity and class diagrams, such as a hyperlink from the object node to the diagram, or insertion of the class diagram in the activity diagram as desired. See example in Figure 12.127.

Examples

See examples at ObjectFlow and children of ObjectNode.

Rationale

Object nodes are introduced to model the flow of objects in an activity.

Changes from previous UML

ObjectNode replaces and extends ObjectFlowState in UML 1.5. In particular, it and its children support collection of tokens at runtime, single sending and receipt, and the new “pin” style of activity model.

12.3.39 ObjectNodeOrderingKind (from CompleteActivities)

Generalizations

None

Description

ObjectNodeOrderingKind is an enumeration indicating queuing order within a node.

Enumeration Values

- unordered
- ordered
- LIFO
- FIFO

12.3.40 OutputPin

Output pins are object nodes that deliver values to other actions through object flows. See Pin, Action, and ObjectNode for more details.

Attributes

No additional attributes

Associations

No additional associations

Constraints

- [1] Output pins may only have incoming edges when they are on actions that are structured nodes, and these edges may not target a node contained by the structured node.

Semantics

See “OutputPin (from BasicActions)” on page 256.

12.3.41 Parameter (from CompleteActivities)

Parameter is specialized when used with complete activities.

Generalizations

- “Parameter (from Kernel, AssociationClasses)” on page 115

Description

Parameters are extended in complete activities to add support for streaming, exceptions, and parameter sets.

Attributes

- `effect` : `ParameterEffectKind` [0..*] Specifies the effect that the owner of the parameter has on values passed in or out of the parameter.
- `isException` : `Boolean` [1..1] =false Tells whether an output parameter may emit a value to the exclusion of the other outputs.
- `isStream` : `Boolean` [1..1] = false Tells whether an input parameter may accept values while its behavior is executing, or whether an output parameter post values while the behavior is executing.
- `parameterSet` : `ParameterSet` [0..*] The parameter sets containing the parameter. See `ParameterSet`.

Associations

No additional associations

Constraints

- [1] A parameter cannot be a stream and exception at the same time.
- [2] An input parameter cannot be an exception.
- [3] Reentrant behaviors cannot have stream parameters.
- [4] Only in and inout parameters may have a delete effect. Only out, inout, and return parameters may have a create effect.

Semantics

`isException` applies to output parameters. An output posted to an exception excludes outputs from being posted to other data and control outputs of the behavior. A token arriving at an exception output parameter of an activity aborts all flows in the activity. Any objects previously posted to non-stream outputs never leave the activity. Streaming outputs posted before any exception are not affected. Use exception parameters on activities only if it is desired to abort all flows in the activity. For example, if the same execution of an activity is being used for all its invocations, then multiple streams of tokens will be flowing through the same activity. In this case, it is probably not desired to abort all tokens just because one reaches an exception. Arrange for separate executions of the activity to use separate executions of the activity when employing exceptions, so tokens from separate executions will not affect each other.

Streaming parameters give an action access to tokens passed from its invoker while the action is executing. Values for streaming parameters may arrive anytime during the execution of the action, not just at the beginning. Multiple value may arrive on a streaming parameter during a single action execution and be consumed by the action. In effect, streaming parameters give an action access to token flows outside of the action while it is executing. In addition to the execution rules given at Action, these rules also apply to invoking a behavior with streaming parameters:

- All required non-stream inputs must arrive for the behavior to be invoked. If there are only required stream inputs, then at least one must arrive for the behavior to be invoked.
- All required inputs must arrive for the behavior to finish.
- Either all required non-exception outputs must be posted by the time the activity is finished, or one of the exception outputs must be. An activity finishes when all its tokens are in its output parameter nodes. If some output parameter nodes are empty at that time, they are assigned the null token (see “Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” on page 306), and the activity terminates.

The execution rules above provide for the arrival of inputs after a behavior is started and the posting of outputs before a behavior is finished. These are stream inputs and outputs. Multiple stream input and output tokens may be consumed and posted while a behavior is running. Since an activity is a kind of behavior, the above rules apply to invoking an activity, even if the invocation is not from another activity. A reentrant behavior cannot have streaming parameters because there are potentially multiple executions of the behavior going at the same time, and it is ambiguous which execution should receive streaming tokens.

The effect of a parameter is a declaration of the modeler's intent, and does not have execution semantics. The modeler must ensure that the owner of the parameter has the stated effect.

See semantics of Action and ActivityParameterNode. Also, see "MultiplicityElement (from Kernel)" on page 90, which inherits to Parameter. It defines a lower and upper bound on the values passed to parameter at runtime. A lower bound of zero means the parameter is optional. Actions using the parameter may execute without having a value for optional parameters. A lower bound greater than zero means values for the parameter are required to arrive sometime during the execution of the action.

Notation

See notation at Pin and ActivityParameterNode. The notation in class diagrams for exceptions and streaming parameters on operations has the keywords "exception" or "stream" in the property string. See notation for Operation.

Examples

See examples at Pin and ActivityParameterNode.

Rationale

Parameter (in Activities) is extended to support invocation of behaviors by activities.

Changes from previous UML

Parameter (in Activities) is new in UML 2.0.

12.3.42 ParameterEffectKind (from CompleteActivities)

Generalizations

None

Description

The datatype ParameterEffectKind is an enumeration that indicates the effect of a behavior on values passed in or out of its parameters (see "Parameter (from CompleteActivities)" on page 383).

Enumeration Values

- create
- read
- update
- delete

12.3.43 ParameterSet (from CompleteActivities)

A parameter set is an element that provides alternative sets of inputs and outputs that a behavior may use.

Generalizations

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

Description

A parameter set acts as a complete set of inputs and outputs to a behavior, exclusive of other parameter sets on the behavior.

Attributes

No additional attributes

Associations (CompleteActivities)

- condition : Constraint [0..*] Constraint that should be satisfied for the owner of the parameters in an input parameter set to start execution using the values provided for those parameters, or the owner of the parameters in an output parameter set to end execution providing the values for those parameters, if all preconditions and conditions on input parameter sets were satisfied.
- parameter : Parameter [1..*] Parameters in the parameter set.

Constraints

- [1] The parameters in a parameter set must all be inputs or all be outputs of the same parameterized entity, and the parameter set is owned by that entity.
- [2] If a behavior has input parameters that are in a parameter set, then any inputs that are not in a parameter set must be streaming. Same for output parameters.
- [3] Two parameter sets cannot have exactly the same set of parameters.

Semantics

A behavior with input parameter sets can only accept inputs from parameters in one of the sets per execution. A behavior with output parameter sets can only post outputs to the parameters in one of the sets per execution. The same is true for operations with parameter sets. The semantics described at Action and ActivityParameter apply to each set separately. The semantics of conditions of input and output parameter sets is the same as Behavior preconditions and postconditions, respectively, but apply only to the set of parameters specified.

Notation

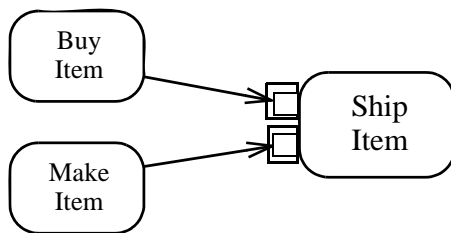
Multiple object flows entering or leaving a behavior invocation are typically treated as “and” conditions. However, sometimes one group of flows are permitted to the exclusion of another. This is modeled as parameter set and notated with rectangles surrounding one or more pins. The notation in the figure below expresses a disjunctive normal form where one group of “and” flows are separated by “or” groupings. For input, when one group or another has a complete set of input flows, the activity may begin. For output, based on the internal processing of the behavior, one group or other of output flows may occur.



Figure 12.117 - Alternative input/outputs using parameter sets notation

Examples

In the figure below, the Ship Item activity begins whenever it receives a bought item or a made item.



Using parameter sets to express “or” invocation

Figure 12.118 - Example of alternative input/outputs using parameter sets

Rationale

Parameter sets provide a way for behaviors to direct token flow in the activity that invokes those behaviors.

Changes from previous UML

ParameterSet is new in UML 2.0.

12.3.44 Pin (from BasicActivities, CompleteActivities)

Generalizations

- “ObjectNode (from BasicActivities, CompleteActivities)” on page 380
- “Pin (from BasicActions)” on page 256 (*merge increment*)

Description

A pin is an object node for inputs and outputs to actions.

Attributes

Package CompleteActivities

- `isControl : Boolean [1..1] = false` Tells whether the pins provide data to the actions, or just controls when it executes it.

Associations

No additional associations

Constraints

See constraints on `ObjectFlow`.

Constraints

Package CompleteActivities

- [1] Control pins have a control type.
`isControl` **implies** `isControlType`

Semantics

See “Pin (from BasicActions)” on page 256.

(CompleteActivities) Control pins always have a control type, so they can be used with control edges. Control pins are ignored in the constraints that actions place on pins, including matching to behavior parameters for actions that invoke behaviors. Tokens arriving at control input pins have the same semantics as control arriving at an action, except that control tokens can queue up in control pins. Tokens are placed on control output pins according to the same semantics as tokens placed on control edges coming out of actions.

Notation

Pin rectangles may be notated as small rectangles that are attached to action rectangles. See figure below and examples. The name of the pin can be displayed near the pin. The name is not restricted, but it often just shows the type of object or data that flows through the pin. It can also be a full specification of the corresponding behavior parameter for invocation actions, using the same notation as parameters for behavioral features on classes. The pins may be elided in the notation even though they are present in the model. Pins that do not correspond to parameters can be labeled as “name:type.”



Figure 12.119 - Pin notations

The situation in which the output pin of one action is connected to the input pin of the same name in another action may be shown by the optional notations of Figure 12.120. The standalone pin in the notation maps to an output pin and an input pin in the underlying model. This form should be avoided if the pins are not of the same type. These variations in notation assume the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements, so that the chosen variation is preserved on interchange.

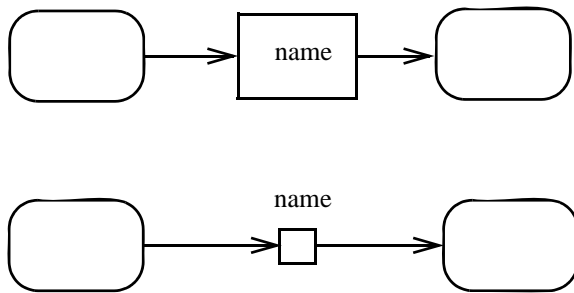


Figure 12.120 - Standalone pin notations

See ObjectNode for other notations applying to pins, with examples for pins below.

Package CompleteActivities

To show streaming, a text annotation is placed near the pin symbol: {stream} or {nonstream}. See figure below. The notation is the same for a standalone object node. Nonstream is the default where the notation is omitted.

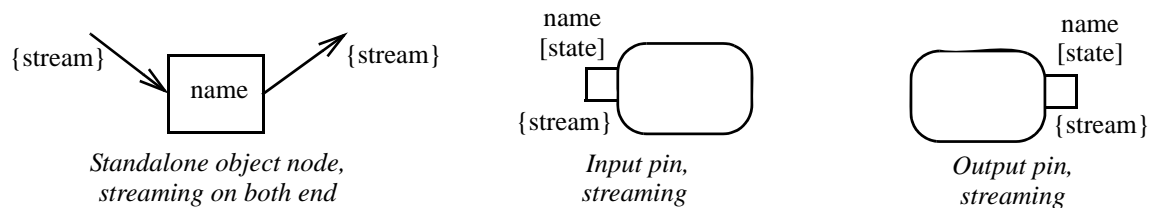


Figure 12.121 - Stream pin notations

Pins for exception parameters are indicated with a small triangle annotating the source end of the edge that comes out of the exception pin. The notation is the same even if the notation uses a standalone notation. See figure below.

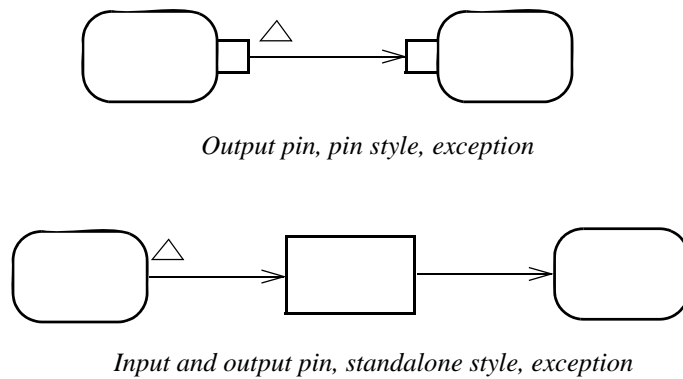


Figure 12.122 - Exception pin notations

Specifying the effect that the behavior of actions has on the objects passed in and out of their parameters can be represented by placing the effect in braces near the edge leading to or from the pin for the parameter.



Figure 12.123 - Specifying effect that actions have on objects

Control pins are shown with a text annotation placed near the pin symbol {control}.

See ObjectNode for other notations applying to pins, with examples for pins below.

Presentation Options

When edges are not present to distinguish input and output pins, an optional arrow may be placed inside the pin rectangle, as shown below. Input pins have the arrow pointing toward the action and output pins have the arrow pointing away from the action.



Figure 12.124 - Pin notations, with arrows

Package CompleteActivities

Additional emphasis may be added to streaming parameters by using a graphical notation instead of the textual adornment. Object nodes can be connected with solid arrows containing filled arrowheads to indicate streaming. Pins can be shown as filled rectangles. When combined with the option above, the arrows are shown as normal arrowheads.

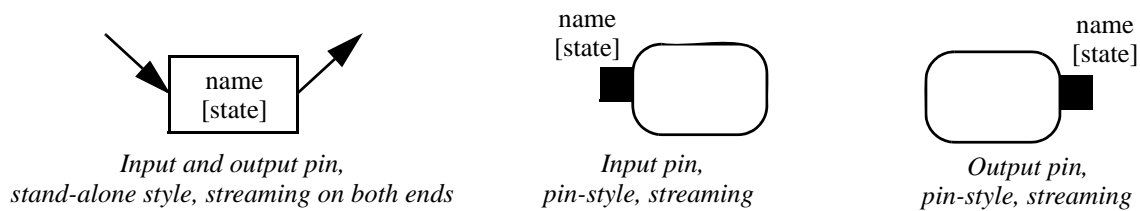


Figure 12.125 - Stream pin notations, with filled arrows and rectangles

Examples

In the example below, the pin named “Order” represents Order objects. In this example at the upper left, the Fill Order behavior produces filled orders and Ship Order consumes them and an invocation of Fill Order must complete for Ship Order to begin. The pin symbols have been elided from the action symbols; both pins are represented by the single box on the arrow. The example on the upper right shows the same thing with explicit pin symbols on actions. The example at the bottom of the figure illustrates the use of multiple pins.

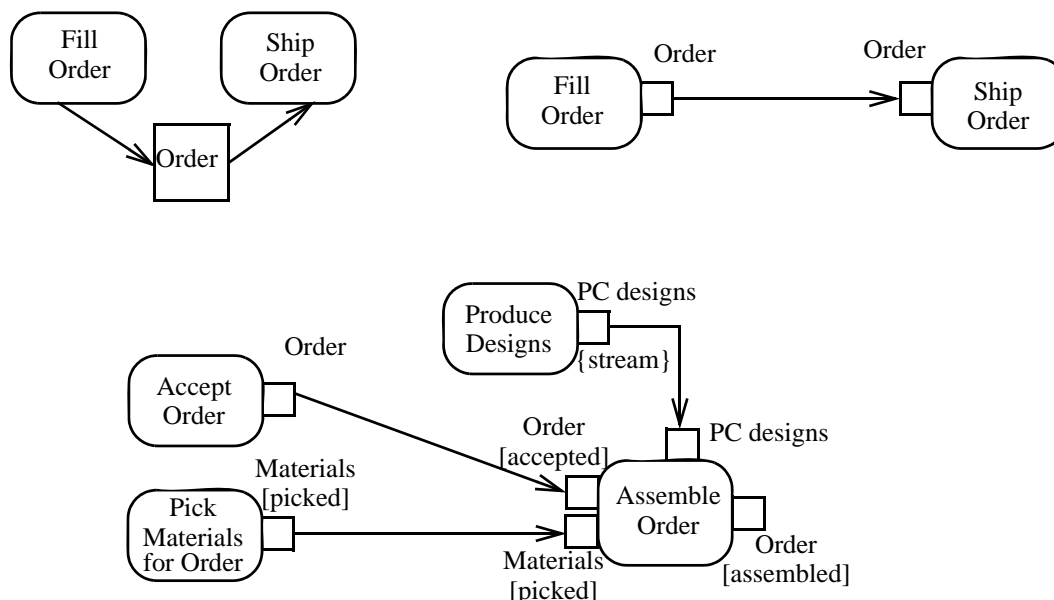


Figure 12.126 - Pin examples

In the figure below, the object node rectangle Order is linked to a class diagram that further defines the node. The class diagram shows that filling an order requires order, line item, and the customer's trim-and-finish requirements. An Order token is the object flowing between the Accept and Fill activities, but linked to other objects. The activity without the class diagram provides a simplified view of the process. The link to an associated class diagram is used to show more detail.

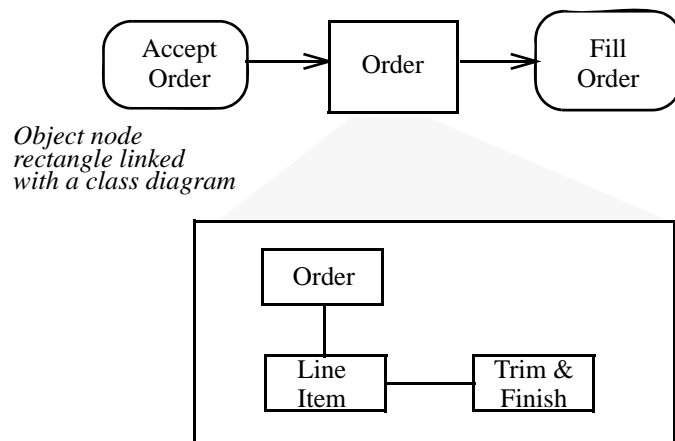


Figure 12.127 - Linking a class diagram to an object node

Package CompleteActivities

In the example below Order Filling is a continuous behavior that periodically emits (streams out) filled-order objects, without necessarily concluding as an activity. The Order Shipping behavior is also a continuous behavior that periodically receives filled-order objects as they are produced. Order Shipping is invoked when the first order arrives and does not terminate, processing orders as they arrive.

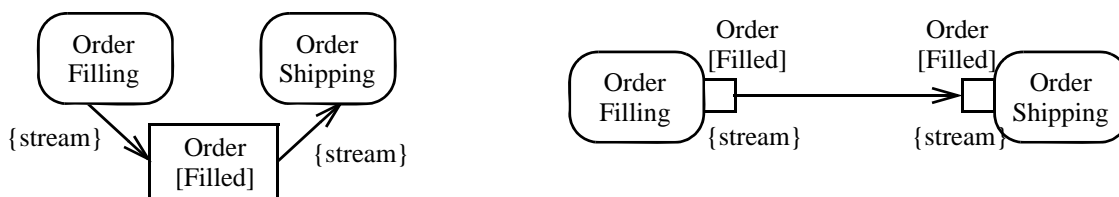


Figure 12.128 - Pin examples

Example of exception notation is shown at the top of the figure below. Accept Payment normally completes with a payment as being accepted and the account is then credited. However, when something goes wrong in the acceptance process, an exception can be raised that the payment is not valid, and the payment is rejected.

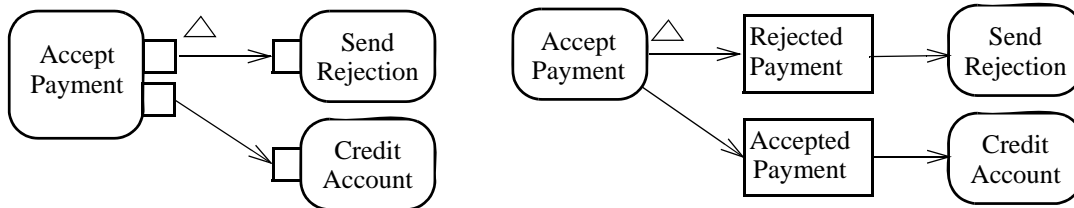


Figure 12.129 - Exception pin examples

The figure below shows two examples of selection behavior. Both examples indicate that orders are to be shipped based on order priority—and those with the same priority should be filled on a first-in/first-out (FIFO) basis.

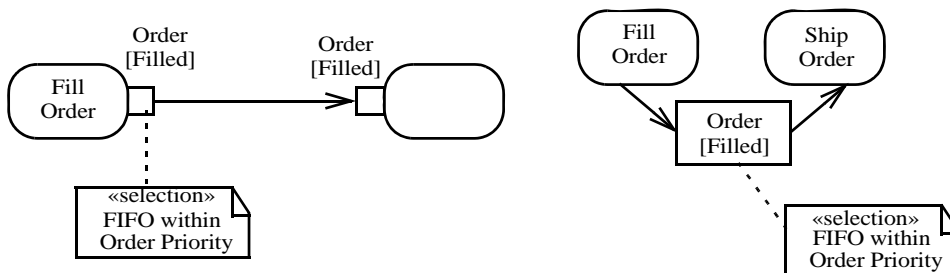


Figure 12.130 - Specifying selection behavior on an ObjectFlow

In the figure below, an example depicts a Place Order activity that creates orders and Fill Order activity that reads these placed orders for the purpose of filling them.

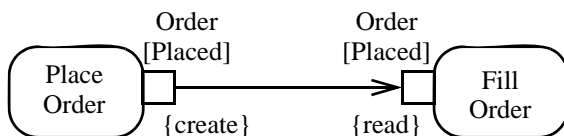


Figure 12.131 Pin example with effects

Rationale

Pin is specialized in Activities to make it an object node and to give it a notation.

Changes from previous UML

Pin is new to activity modeling in UML 2.0. It replaces pins from UML 1.5 action model.

12.3.45 SendObjectAction (as specialized)

See “SendObjectAction (from IntermediateActions)” on page 272.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “SendObjectAction (from IntermediateActions)” on page 272.

Notation

No specific notation

Presentation Options

See “SendObjectAction (from IntermediateActions)” on page 272.

Changes from previous UML

See “SendObjectAction (from IntermediateActions)” on page 272.

12.3.46 SendSignalAction (as specialized)

See “SendSignalAction (from BasicActions)” on page 273.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “SendSignalAction (from BasicActions)” on page 273.

Notation

See “SendSignalAction (from BasicActions)” on page 273.

Examples

Figure 12.132 shows part of an order-processing workflow in which two signals are sent. An order is created (in response to some previous request that is not shown in the example). A signal is sent to the warehouse to fill and ship the order. Then an invoice is created and sent to the customer.

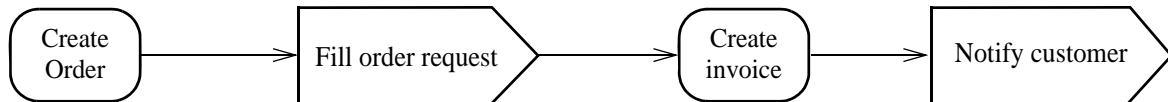


Figure 12.132 - Signal node notations

Rationale

See “SendSignalAction (from BasicActions)” on page 273.

Changes from previous UML

See “SendSignalAction (from BasicActions)” on page 273.

12.3.47 SequenceNode (from StructuredActivities)

Generalizations

- “StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)” on page 396

Description

(StructuredActivities) A sequence node is a structured activity node that executes its actions in order.

Attributes

No additional attributes

Associations

- executableNode : ExecutableNode [*] An ordered set of executable nodes.

Constraints

No additional constraints

Semantics

When the sequence node is enabled, its executable nodes are executed in the order specified. When combined with flows, actions must also satisfy their control and data flow inputs before starting execution.

Notation

No specific notation

Rationale

SequenceNode is introduced to provide a way for structured activities to model a sequence of actions.

Changes from previous UML

SequenceNode is new to UML 2.

12.3.48 StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)

(StructuredActivities) A structured activity node is an executable activity node that may have an expansion into subordinate nodes as an ActivityGroup. The subordinate nodes must belong to only one structured activity node, although they may be nested.

Generalizations

- “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” on page 301
- “ActivityGroup (from BasicActivities, FundamentalActivities)” on page 322
- “ExecutableNode (from ExtraStructuredActivities, StructuredActivities)” on page 354
- “Namespace (from Kernel)” on page 95

Description

A structured activity node represents a structured portion of the activity that is not shared with any other structured node, except for nesting. It may have control edges connected to it, and pins in CompleteStructuredActivities. The execution of any embedded actions may not begin until the structured activity node has received its object and control tokens. The availability of output tokens from the structured activity node does not occur until all embedded actions have completed execution (see exception at AcceptEventAction (from CompleteActions)).

Package CompleteStructuredActivities

Because of the concurrent nature of the execution of actions within and across activities, it can be difficult to guarantee the consistent access and modification of object memory. In order to avoid race conditions or other concurrency-related problems, it is sometimes necessary to isolate the effects of a group of actions from the effects of actions outside the group. This may be indicated by setting the mustIsolate attribute to *true* on a structured activity node. If a structured activity node is “isolated,” then any object used by an action within the node cannot be accessed by any action outside the node until the structured activity node as a whole completes. Any concurrent actions that would result in accessing such objects are required to have their execution deferred until the completion of the node.

Note – Any required isolation may be achieved using a locking mechanism, or it may simply sequentialize execution to avoid concurrency conflicts. Isolation is different from the property of “atomicity,” which is the guarantee that a group of actions either all complete successfully or have no effect at all. Atomicity generally requires a rollback mechanism to prevent committing partial results.

Attributes

- `mustIsolate` : Boolean If *true*, then the actions in the node execute in isolation from actions outside the node.

Associations

- variable: Variable [0..*] A variable defined in the scope of the structured activity node. It has no value and may not be accessed outside the node.

Package StructuredActivities

- containedNode : ActivityNode [0..*] Nodes immediately contained in the group. (Redefines *ActivityGroup::containedNode*.)

Constraints

Package CompleteStructuredActivities

[1] The edges owned by a structured node must have source and target nodes in the structured node, and vice versa.

Semantics

Nodes and edges contained by a structured node cannot be contained by any other structured node. This constraint is modeled as a specialized multiplicity from ActivityNode and ActivityEdge to StructuredActivityNode. Edges not contained by a structured node can have sources or targets in the structured node, but not both. See children of StructuredActivityNode.

No subnode in the structured node, including initial nodes and accept event actions, may begin execution until the structured node itself has started. Subnodes begin executing according to the same rules as the subnodes of an activity (see “InitialNode (from BasicActivities)” on page 365 and “AcceptEventAction (from CompleteActions)” on page 228). A control flow from a structured activity node implies that a token is produced on the flow only after no tokens are left in the node or its contained nodes recursively. Tokens reaching an activity final node in a structured node abort all flows in the immediately containing structured node only. The other aspects of termination are the same as for activity finals contained directly by activities (see “ActivityFinalNode (from BasicActivities, IntermediateActivities)” on page 320).

Package CompleteStructuredActivities

An object node attached to a structured activity node is accessible within the node. The same rules apply as for control flow. Input pins on a structured activity node imply that actions in the node begin execution when all input pins have received tokens. An output pin on a structured activity node will make tokens available outside the node only after no tokens are left in the node or its contained nodes recursively.

If the mustIsolate flag is true for an activity node, then any access to an object by an action within the node must not conflict with access to the object by an action outside the node. A conflict is defined as an attempt to write to the object by one or both of the actions. If such a conflict potentially exists, then no such access by an action outside the node may be interleaved with the execution of any action inside the node. This specification does not constrain the ways in which this rule may be enforced. If it is impossible to execute a model in accordance with these rules, then it is ill formed.

Notation

A structured activity node is notated with a dashed round cornered rectangle enclosed its nodes and edges, with the keyword «structured» at the top. Also see children of StructuredActivityNode.

Examples

See children of StructuredActivityNode.

Rationale

StructuredActivityNode is for applications that require well-nested nodes. It provides well-nested nodes that were enforced by strict nesting rules in UML 1.5.

Changes from previous UML

StructuredActivityNode is new in UML 2.0.

12.3.49 UnmarshallAction (as specialized)

See “UnmarshallAction (from CompleteActions)” on page 278.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “UnmarshallAction (from CompleteActions)” on page 278.

Notation

No specific notation

Examples

In Figure 12.133, an order is unmarshalled into its name, shipping address, and product.

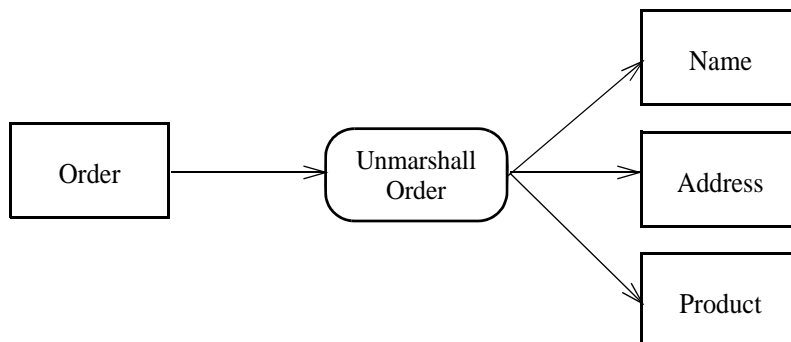


Figure 12.133 - Example of UnmarshallAction

Rationale

See “UnmarshallAction (from CompleteActions)” on page 278.

Changes from previous UML

See “UnmarshallAction (from CompleteActions)” on page 278.

12.3.50 ValuePin (as specialized)

A value pin is an input pin that provides a value to an action that does not come from an incoming object flow edge.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] Value pins have no incoming edges.

Semantics

ValuePins provide values to their actions, but only when the actions are otherwise enabled. If an action has no incoming edges or other way to start execution, a value pin will not start the execution by itself or collect tokens waiting for execution to start. When the action is enabled by these other means, the value specification of the value pin is evaluated and the result provided as input to the action, which begins execution. This is an exception to the normal token flow semantics of activities.

Notation

A value pin is notated as an input pin with the value specification written beside it.

Rationale

ValuePin is introduced to reduce the size of activity models that use constant values. See “ValueSpecificationAction (from IntermediateActions)” on page 280.

Changes from UML 1.5

ValuePin replaces LiteralValueAction from UML 1.5.

12.3.51 ValueSpecificationAction (as specialized)

See “ValueSpecificationAction (from IntermediateActions)” on page 280.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “ValueSpecificationAction (from IntermediateActions)” on page 280.

Notation

The action is labeled with the value specification, as shown in Figure 12.134.

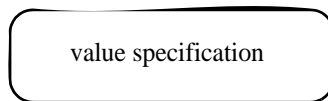


Figure 12.134 - ValueSpecificationAction notation

Examples

Figure 12.135 shows a value specification action used to output a constant from an activity.

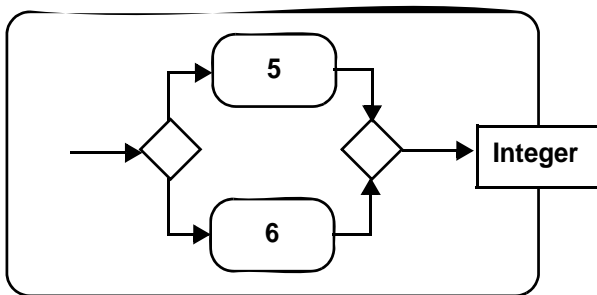


Figure 12.135 - Example ValueSpecificationAction

Rationale

See “ValueSpecificationAction (from IntermediateActions)” on page 280.

Changes from previous UML

See “ValueSpecificationAction (from IntermediateActions)” on page 280.

12.3.52 Variable (from StructuredActivities)

(StructuredActivities) Variables are elements for passing data between actions indirectly. A local variable stores values shared by the actions within a structured activity group but not accessible outside it. The output of an action may be written to a variable and read for the input to a subsequent action, which is effectively an indirect data flow path. Because there is no predefined relationship between actions that read and write variables, these actions must be sequenced by control flows to prevent race conditions that may occur between actions that read or write the same variable.

Generalizations

- “MultiplicityElement (from Kernel)” on page 90
- “TypedElement (from Kernel)” on page 131

Description

A variable specifies data storage shared by the actions within a group. There are actions to write and read variables. These actions are treated as side effecting actions, similar to the actions to write and read object attributes and associations. There are no sequencing constraints among actions that access the same variable. Such actions must be explicitly coordinated by control flows or other constraints.

Any values contained by a variable must conform to the type of the variable and have cardinalities allowed by the multiplicity of the variable.

Associations

- scope : StructuredActivityNode [0..1] A structured activity node that owns the variable.
- activityScope : Activity [0..1] An activity that owns the variable.

Attributes

No additional attributes

Constraints

[1] A variable is owned by a StructuredNode or Activity, but not both.

Additional operations

[1] The isAccessibleBy() operation is not defined in standard UML. Implementations should define it to specify which actions can access a variable.

isAccessibleBy(a: Action) : Boolean

Semantics

A variable specifies a slot able to hold a value or a sequence of values, consistent with the multiplicity of the variable. The values held in this slot may be accessed from any action contained directly or indirectly within the group action or activity that is the scope of the variable.

Notation

No specific notation

Rationale

Variables are introduced to simplify translation of common programming languages into activity models for those applications that do not require object flow information to be readily accessible. However, source programs that set variables only once can be easily translated to use object flows from the action that determines the values to the actions that use them. Source programs that set variables more than once can be translated to object flows by introducing a local object containing attributes for the variables, or one object per variable combined with data store nodes.

Changes from UML 1.5

Variable is unchanged from UML 1.5, except that it is used on `StructuredActivityNode` instead of `GroupNode`.

12.4 Diagrams

The focus of activity modeling is the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The behaviors coordinated by these models can be initiated because other behaviors finish executing, because objects and data become available, or because events occur external to the flow. See 12.3.4, “Activity (from `BasicActivities`, `CompleteActivities`, `FundamentalActivities`, `StructuredActivities`),” on page 306 for more introduction and semantic framework.

The notation for activities is optional. A textual notation may be used instead.

The following sections describe the graphic nodes and paths that may be shown in activity diagrams.

Graphic Nodes

The graphic nodes that can be included in activity diagrams are shown in Table 12.1.

Table 12.1 - Graphic nodes included in activity diagrams

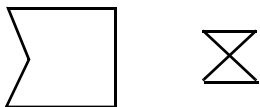


Node Type	Notation	Reference
AcceptEventAction		See “AcceptEventAction (as specialized)” on page 299.
Action		See “Action (from <code>CompleteActivities</code> , <code>FundamentalActivities</code> , <code>StructuredActivities</code>)” on page 301.
ActivityFinal		See “ActivityFinalNode (from <code>BasicActivities</code> , <code>IntermediateActivities</code>)” on page 320.
ActivityNode	<i>See ExecutableNode, ControlNode, and ObjectNode.</i>	See “ActivityNode (from <code>BasicActivities</code> , <code>CompleteActivities</code> , <code>FundamentalActivities</code> , <code>IntermediateActivities</code> , <code>StructuredActivities</code>)” on page 323.

Table 12.1 - Graphic nodes included in activity diagrams

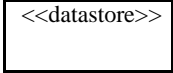


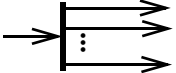

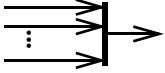
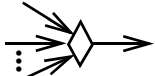
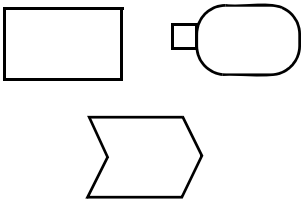
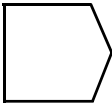
Node Type	Notation	Reference
ControlNode	<i>See DecisionNode, FinalNode, ForkNode, InitialNode, JoinNode, and MergeNode.</i>	See “ControlNode (from BasicActivities)” on page 346.
DataStore		See “DataStoreNode (from CompleteActivities)” on page 347.
DecisionNode		See “DecisionNode (from IntermediateActivities)” on page 349.
FinalNode	<i>See ActivityFinal and FlowFinal.</i>	See “FinalNode (from IntermediateActivities)” on page 360.
FlowFinal		See “FlowFinalNode (from IntermediateActivities)” on page 362.
ForkNode		See “ForkNode (from IntermediateActivities)” on page 363.
InitialNode		See “InitialNode (from BasicActivities)” on page 365.
JoinNode		See “JoinNode (from CompleteActivities, IntermediateActivities)” on page 368.
MergeNode		See “MergeNode (from IntermediateActivities)” on page 373.

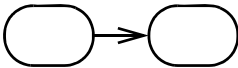
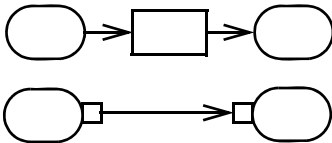
Table 12.1 - Graphic nodes included in activity diagrams

Node Type	Notation	Reference
ObjectNode		See “ObjectNode (from BasicActivities, CompleteActivities)” on page 380 and its children.
SendSignalAction		See “SendSignalAction (as specialized)” on page 394.

Graphic Paths

The graphic paths that can be included in activity diagrams are shown in Table 12.2

Table 12.2 - Graphic paths included in activity diagrams

Path Type		Reference
ActivityEdge	<i>See ControlFlow and ObjectFlow.</i>	See “ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)” on page 315.
ControlFlow		See “ControlFlow (from BasicActivities)” on page 344.
ObjectFlow		See “ObjectFlow (from BasicActivities, CompleteActivities)” on page 375 and its children.

Other Graphical Elements

Activity diagrams have graphical elements for containment. These are included in Table 12.3.

Table 12.3 - Graphic elements for containment in activity diagrams

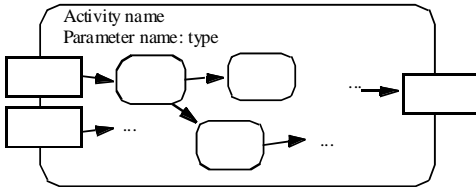
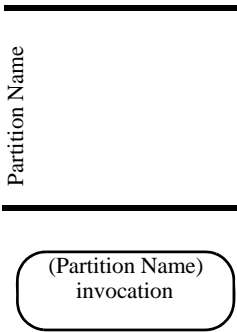
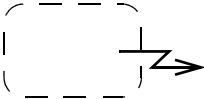
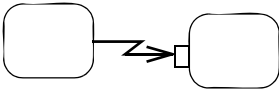
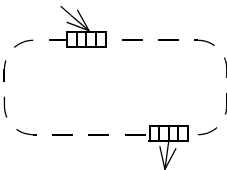
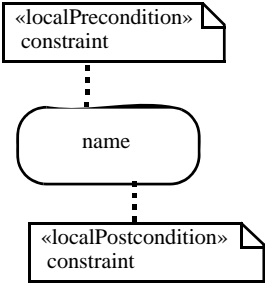
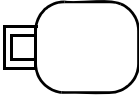
Type	Notation	Reference
Activity		See “Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” on page 306.
ActivityPartition		See “ActivityPartition (from IntermediateActivities)” on page 329.
InterruptibleActivityRegion		See “InterruptibleActivityRegion (from CompleteActivities)” on page 366.
ExceptionHandler		See “ExceptionHandler (from ExtraStructuredActivities)” on page 351.
ExpansionRegion		“ExpansionRegion (from ExtraStructuredActivities)” on page 355

Table 12.3 - Graphic elements for containment in activity diagrams

Type	Notation	Reference
Local pre- and postconditions.		See “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” on page 301.
ParameterSet		See “ParameterSet (from CompleteActivities)” on page 386.

13 Common Behaviors

13.1 Overview

The Common Behaviors packages specify the core concepts required for dynamic elements and provides the infrastructure to support more detailed definitions of behavior. Figure 13.1 shows a domain model explaining the relationship between occurrences of behaviors.

Note – The models shown in Figure 13.1 through Figure 13.4 are not metamodels but show objects in the semantic domain and relationships between these objects. These models are used to give an informal explication of the dynamic semantics of the classes of the UML metamodel.

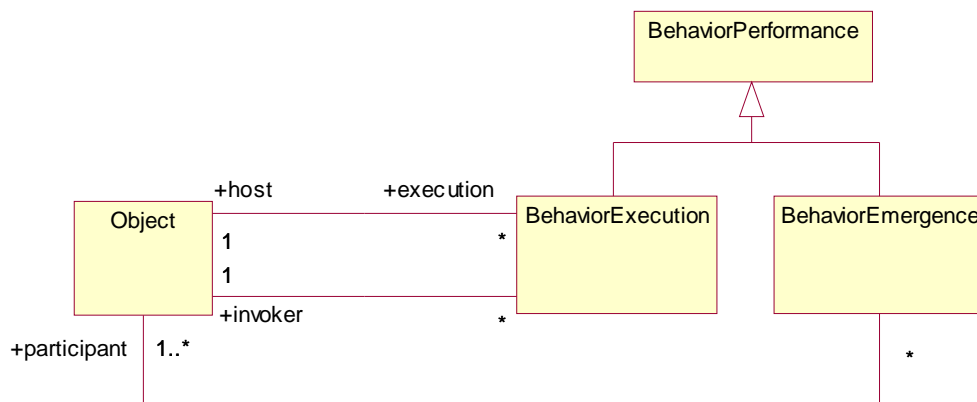


Figure 13.1 - Common Behaviors Domain Model

Any behavior is the direct consequence of the action of at least one object. A behavior describes how the states of these objects, as reflected by their structural features, change over time. Behaviors, as such, do not exist on their own, and they do not communicate. If a behavior operates on data, that data is obtained from the host object.

There are two kinds of behaviors, emergent behavior and executing behavior. An *executing behavior* is performed by an object (its host) and is the description of the behavior of this object. An executing behavior is directly caused by the invocation of a behavioral feature of that object or by its creation. In either case, it is a consequence of the execution of an action by some related object. A behavior has access to the structural features of its host object. Objects that may host behaviors are specified by the concrete subtypes of the *BehavioredClassifier* metaclass.

Emergent behavior results from the interaction of one or more participant objects. If the participating objects are parts of a larger composite object, an emerging behavior can be seen as indirectly describing the behavior of the container object also. Nevertheless, an emergent behavior can result from the executing behaviors of the participant objects.

Occurring behaviors are specified by the concrete subtypes of the abstract *Behavior* metaclass. Behavior specifications can be used to define the behavior of an object, or they can be used to describe or illustrate the behavior of an object. The latter may only focus on a relevant subset of the behavior an object may exhibit (allowed behavior), or it may focus on behavior an object must not exhibit (forbidden behavior).

Albeit behavior is ultimately related to an object, emergent behavior may also be specified for non-instantiable classifiers, such as interfaces or collaborations. Such behaviors describe the interaction of the objects that realize the interfaces or the parts of the collaboration (see “Collaboration (from Collaborations)” on page 164).

BasicBehaviors

The BasicBehaviors subpackage of the Common Behavior package introduces the framework that will be used to specify behaviors. The concrete subtypes of Behavior will provide different mechanisms to specify behaviors. A variety of specification mechanisms are supported by the UML, such as automata (“StateMachine (from BehaviorStateMachines)” on page 545), Petri-net like graphs (“Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” on page 306), informal descriptions (“UseCase (from UseCases)” on page 578), or partially-ordered sequences of event occurrences (“Interaction (from BasicInteraction, Fragments)” on page 467). Profiles may introduce additional styles of behavioral specification. The styles of behavioral specification differ in their expressive power and domain of applicability. Further, they may specify behaviors either explicitly, by describing the observable event occurrences resulting from the execution of the behavior, or implicitly, by describing a machine that would induce these events. The relationship between a specified behavior and its hosting or participating instances is independent of the specification mechanism chosen and described in the common behavior package. The choice of specification mechanism is one of convenience and purpose; typically, the same kind of behavior could be described by any of the different mechanisms. Note that not all behaviors can be described by each of the different specification mechanisms, as these do not all have the same expressive power. However, for many behaviors, the choice of specification mechanism is one of convenience.

As shown in the domain model of Figure 13.2, the execution of a behavior may be caused by a call behavior occurrence (representing the direct invocation of a behavior through an action) or a trigger occurrence (representing an indirect invocation of a behavior, such as through an operation call). A start occurrence marks the beginning of a behavior execution, while its completion is accompanied by a termination occurrence.

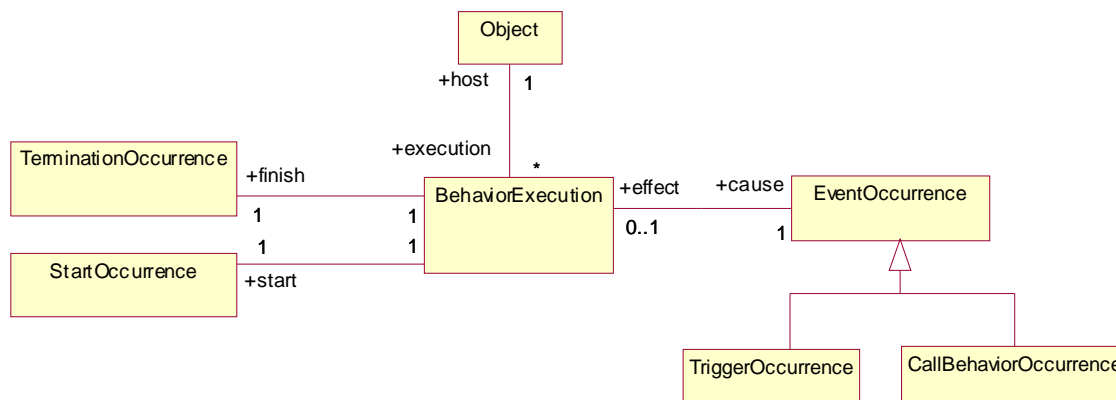


Figure 13.2 - Invocation Domain Model

Communications

The Communications subpackage of the Common Behavior package adds the infrastructure to communicate between objects in the system and to invoke behaviors. The domain model shown in Figure 13.3 explains how communication takes place. Note that this domain model specifies the semantics of communication between objects in a system. Not all aspects of the domain model are explicitly represented in the specification of the system, but may be implied by the dynamic semantics of the constructs used in a specification.

An action representing the invocation of a behavioral feature is executed by a sender object resulting in an invocation event occurring. The invocation event may represent the sending of a signal or the call to an operation. As a result of the invocation event a request is generated. A request is an object capturing the data that was passed to the action causing the invocation event (the arguments that must match the parameters of the invoked behavioral feature); information about the nature of the request (i.e., the behavioral feature that was invoked); the identities of the sender and receiver objects; as well as sufficient information about the behavior execution to enable the return of a reply from the invoked behavior, where appropriate. (In profiles, the request object may include additional information, for example, a time stamp.)

While each request is targeted at exactly one receiver object and caused by exactly one sending object, an occurrence of an invocation event may result in a number of requests being generated (as in a signal broadcast). The receiver may be the same object that is the sender, it may be local (i.e., an object held in a slot of the currently executing object, or the currently executing object itself, or the object owning the currently executing object), or it may be remote. The manner of transmitting the request object, the amount of time required to transmit it, the order in which the transmissions reach their receiver objects, and the path for reaching the receiver objects are undefined. Once the generated request arrives at the receiver object, a receiving event will occur.

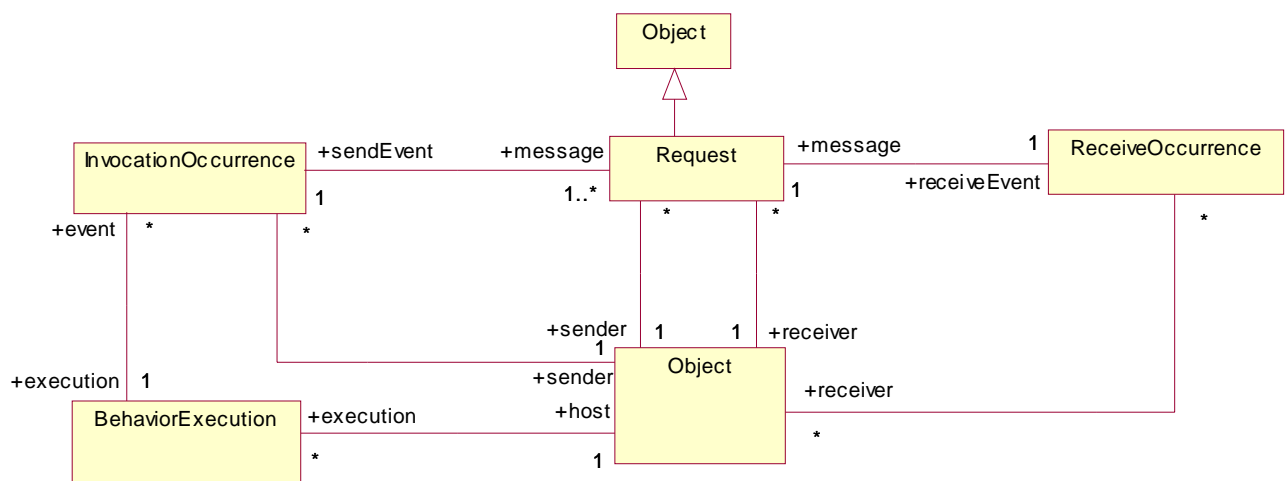


Figure 13.3 - Communication Domain Model

Several kinds of requests exist between instances, for example, sending a signal or invoking an operation. The kind of request is determined by the kind of invocation occurrence that caused it, as shown in Figure 13.4. The former is used to trigger a reaction in the receiver in an asynchronous way and without a reply, while the latter applies an operation to an instance, which may be either synchronously or asynchronously and may require a reply from the receiver to the sender. A send invocation occurrence creates a send request and causes a signal occurrence in the receiver. A call invocation occurrence creates a call request and causes a call occurrence in the receiver.

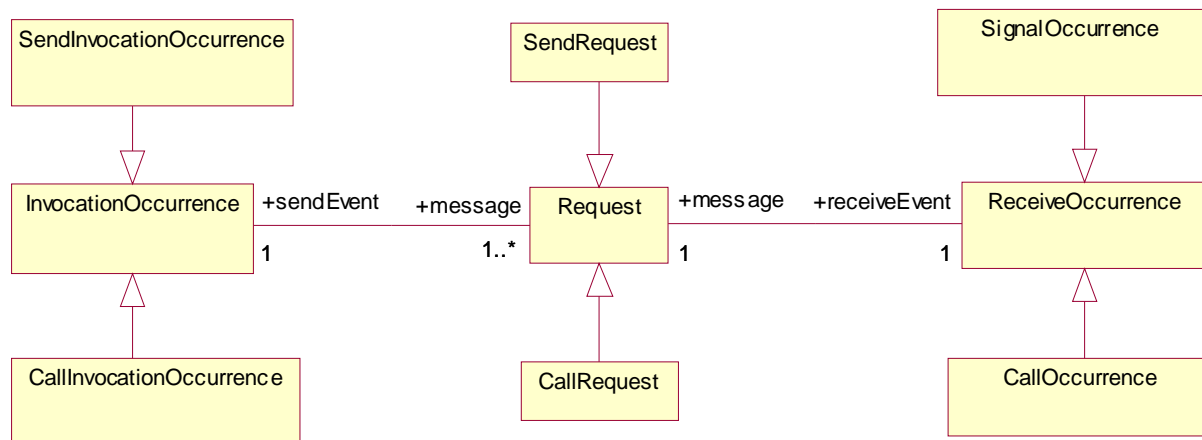


Figure 13.4 - Domain Model Showing Request Kinds

An invocation event occurrence represents the recognition of an invocation request after its receipt by a target object. Invocation event occurrences are the result of the execution of invocation actions (see “InvocationAction (from BasicActions)” on page 249). Invocation actions include send actions and call actions. A send action is specified by a Signal (see “Signal (from Communications)” on page 435) and argument values. The execution of a send action results in a send request, which results in a call event occurrence when it is recognized by the target object. A call action is specified by an Operation and argument values. The execution of a call action results in a call request, which results in a call event occurrence when it is recognized by the target object. Signal event occurrences and call event occurrences are specified by the corresponding metaclasses (see “SignalEvent (from Communications)” on page 435 and “CallEvent (from Communications)” on page 421).

As shown in Figure 13.3, an object hosts a behavior execution (i.e., a behavior will be executed in the context of that object). The execution of an invocation action by the behavior constitutes an invocation occurrence. The invocation occurrence results in a request object that transmits the invocation request from the sender object (caller) to the receiver object (target). The receipt of the request by the receiver is manifest as a receive occurrence. When the receive occurrence matches a trigger defined in the class of the target object, it causes the execution of a behavior. The details of identifying the behavior to be invoked in response to the occurrence of an event are a semantic variation point. The resulting behavior execution is hosted by the target object. The specific mechanism by which the data passed with the request (the attributes of the request object) are made available as arguments to the invoked behavior (e.g., whether the data or copies are passed with the request) is a semantic variation point. If the invocation action is synchronous, the request object also includes sufficient information to identify the execution that invoked the behavior, but this information is not available for the use of the invoked behavior (and, therefore, is not modeled). When a synchronous execution completes, this information is used to direct a reply message to the original behavior execution.

The detection of an (event) occurrence by an object may cause a behavioral response. For example, a state machine may transition to a new state upon the detection of the occurrence of an event specified by a trigger owned by the state machine, or an activity may be enabled upon the receipt of a message. When an event occurrence is recognized by an object, it may have an immediate effect or the event may be saved in an event pool and have a later effect when it is matched by a trigger specified for a behavior.

The occurrence of a change event (see “ChangeEvent (from Communications)” on page 422) is based on some expression becoming true. A time event occurs when a predetermined deadline expires (see “TimeEvent (from Communications, SimpleTime)” on page 438). No data is passed by the occurrence of a change event or a time event. Figure 13.12 shows the hierarchy of events.

SimpleTime

The SimpleTime subpackage of the Common Behavior package adds metaclasses to represent time and durations, as well as actions to observe the passing of time.

The simple model of time described here is intended as an approximation for situations where the more complex aspects of time and time measurement can safely be ignored. For example, this model does not account for the relativistic effects that occur in many distributed systems, or the effects resulting from imperfect clocks with finite resolution, overflows, drift, skew, etc. It is assumed that applications for which such characteristics are relevant will use a more sophisticated model of time provided by an appropriate profile.

13.2 Abstract syntax

Figure 13.5 shows the dependencies of the CommonBehaviors packages.

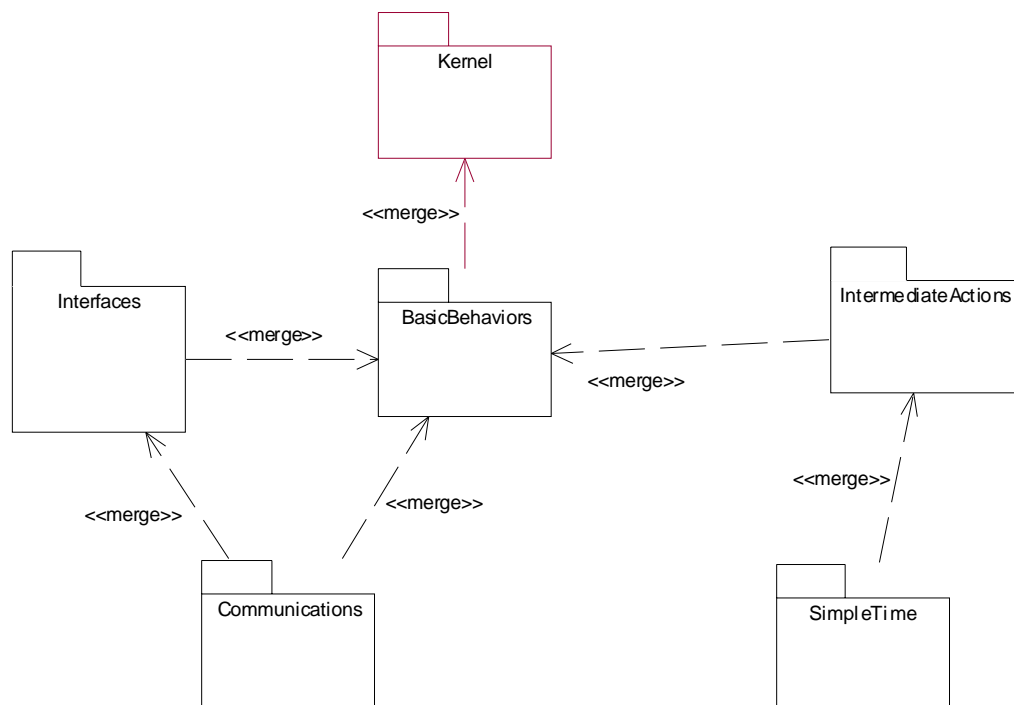


Figure 13.5 - Dependencies of the CommonBehaviors packages

BasicBehaviors

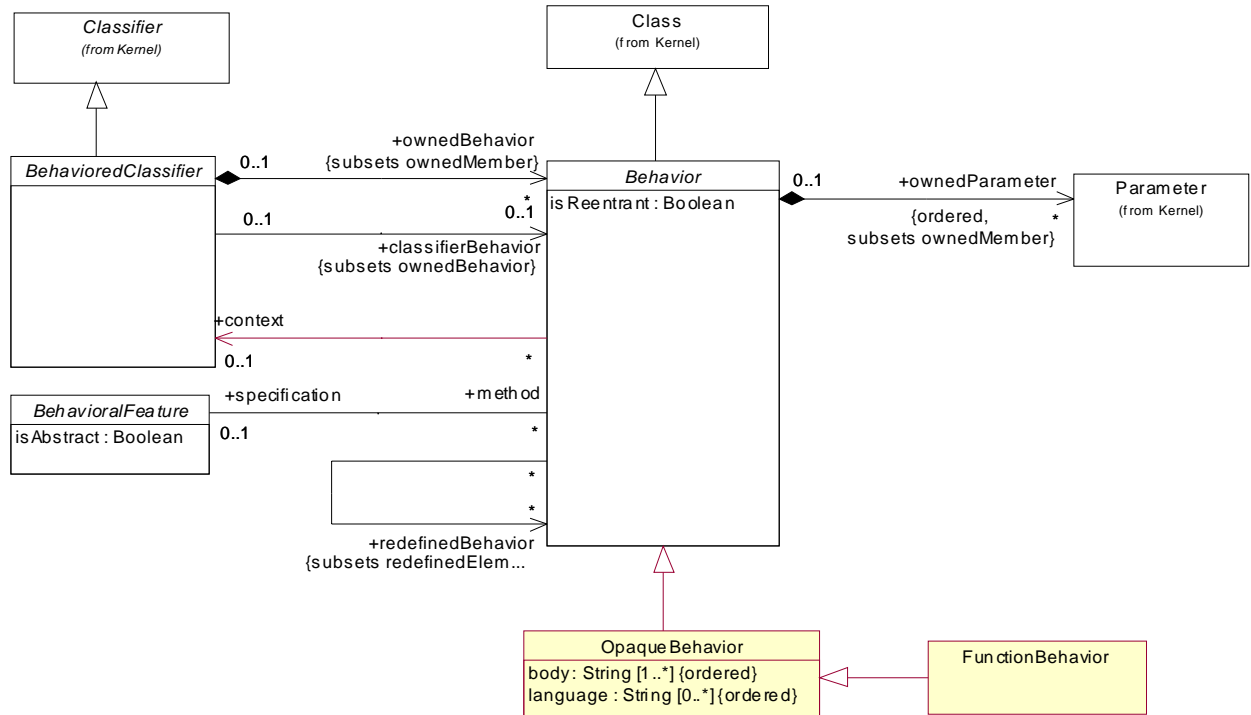


Figure 13.6 - Common Behavior

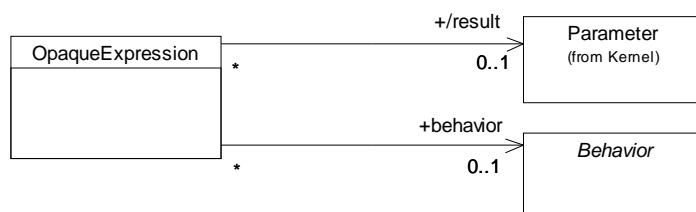


Figure 13.7 - Expression

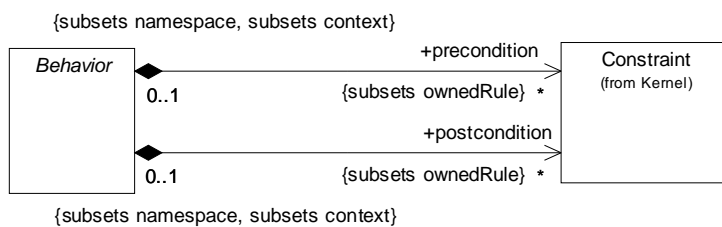


Figure 13.8 - Precondition and postcondition constraints for behavior

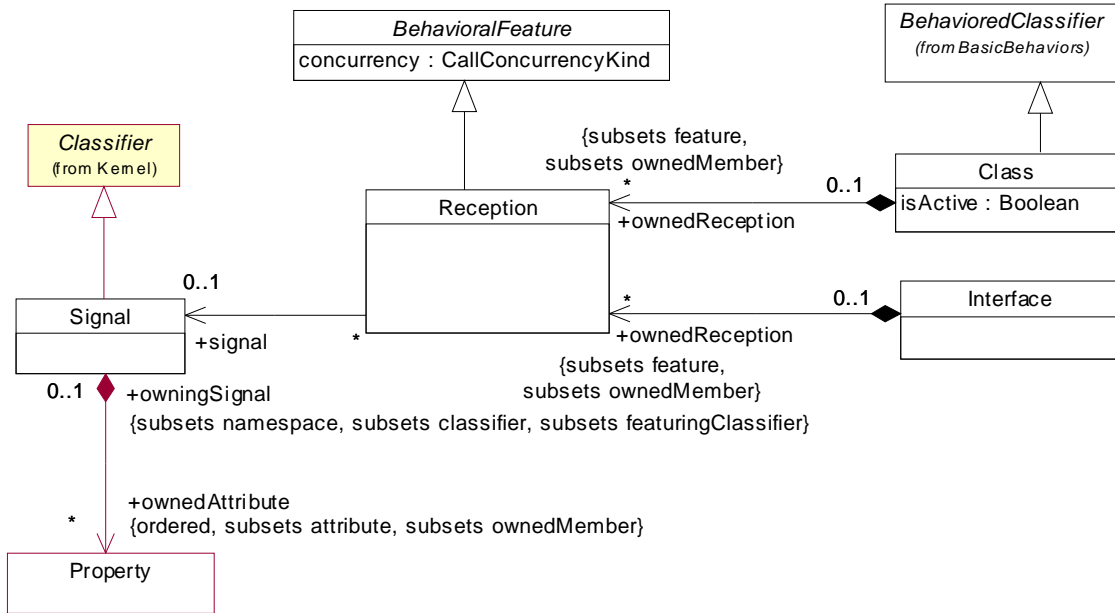


Figure 13.9 - Reception

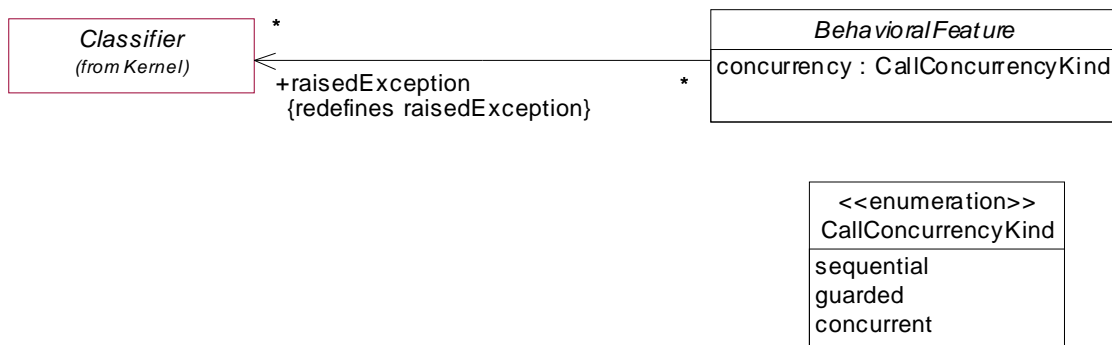


Figure 13.10 - Extensions to behavioral features

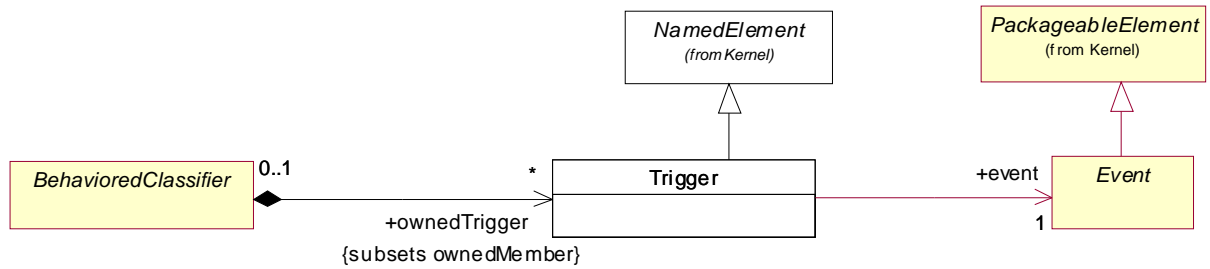


Figure 13.11 - Triggers

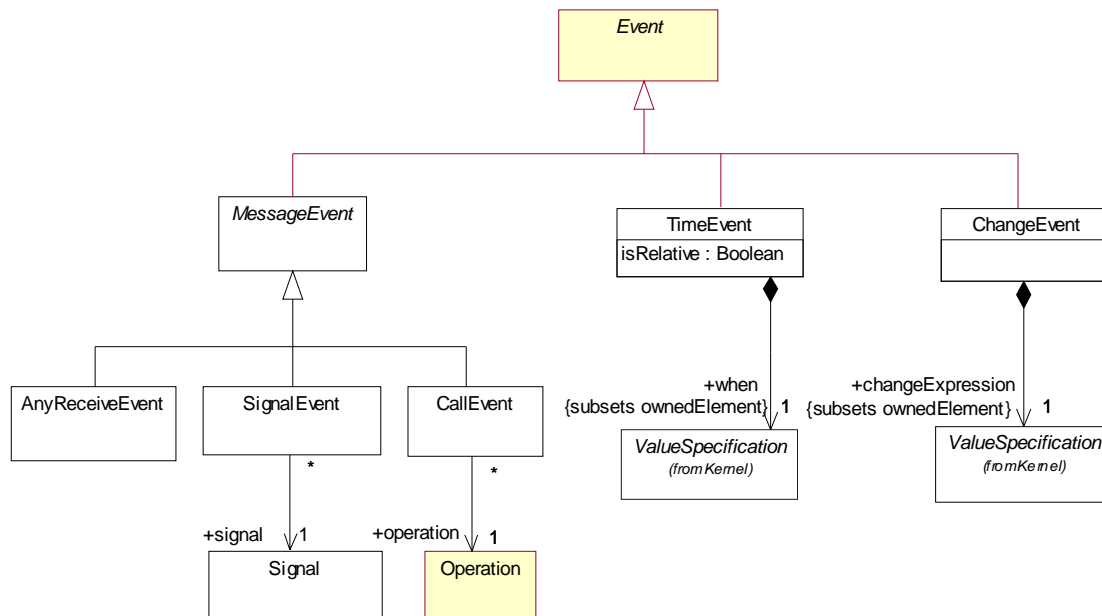


Figure 13.12 - Events

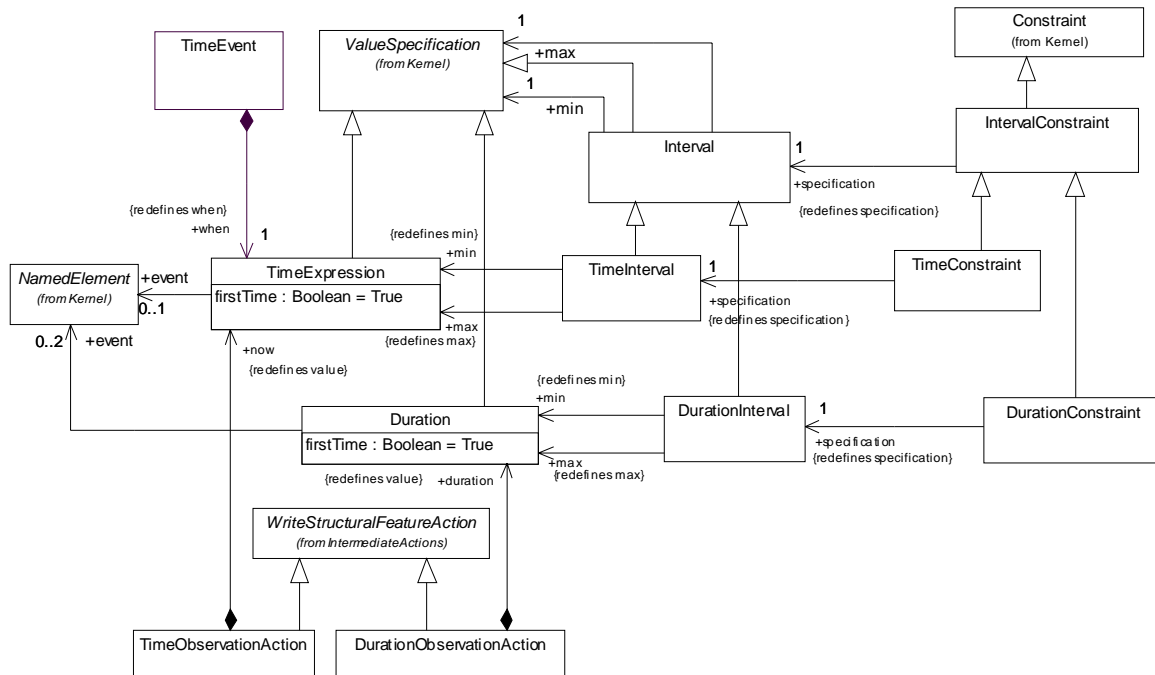


Figure 13.13 - SimpleTime

13.3 Class Descriptions

13.3.1 AnyReceiveEvent (from Communications)

Generalizations

- “MessageEvent (from Communications)” on page 431

Description

A transition trigger associated with AnyReceiveEvent specifies that the transition is to be triggered by the receipt of any message that is not explicitly referenced in another transition from the same vertex.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An AnyReceiveEvent associated with a transition trigger specifies that the transition is triggered for all applicable message receive events except for those specified explicitly on other transitions having the same vertex as a source.

Notation

Any AnyReceiveEvent is denoted by the string “all” used as the trigger.

<any-receive-event> ::= ‘all’

Changes from previous UML

This construct has been added.

13.3.2 Behavior (from BasicBehaviors)

Generalizations

- “Class (from Kernel)” on page 45

Description

Behavior is a specification of how its context classifier changes state over time. This specification may be either a definition of possible behavior execution or emergent behavior, or a selective illustration of an interesting subset of possible executions. The latter form is typically used for capturing examples, such as a trace of a particular execution.

A classifier behavior is always a definition of behavior and not an illustration. It describes the sequence of state changes an instance of a classifier may undergo in the course of its lifetime. Its precise semantics depends on the kind of classifier. For example, the classifier behavior of a collaboration represents emergent behavior of all the parts, whereas the classifier behavior of a class is just the behavior of instances of the class separated from the behaviors of any of its parts.

When a behavior is associated as the method of a behavioral feature, it defines the implementation of that feature (i.e., the computation that generates the effects of the behavioral feature).

As a classifier, a behavior can be specialized. Instantiating a behavior is referred to as “invoking” the behavior, an instantiated behavior is also called a behavior “execution.” A behavior may be invoked directly or its invocation may be the result of invoking the behavioral feature that specifies this behavior. A behavior can also be instantiated as an object in virtue of it being a class.

The specification of a behavior can take a number of forms, as described in the subclasses of Behavior. Behavior is an abstract metaclass factoring out the commonalities of these different specification mechanisms.

When a behavior is invoked, its execution receives a set of input values that are used to affect the course of execution, and as a result of its execution it produces a set of output values that are returned, as specified by its parameters. The observable effects of a behavior execution may include changes of values of various objects involved in the execution, the creation and destruction of objects, generation of communications between objects, as well as an explicit set of output values.

Attributes

- **isReentrant:** Boolean [1] Tells whether the behavior can be invoked while it is still executing from a previous invocation.

Associations

- **specification:** BehavioralFeature [0..1]
Designates a behavioral feature that the behavior implements. The behavioral feature must be owned by the classifier that owns the behavior or be inherited by it. The parameters of the behavioral feature and the implementing behavior must match. If a behavior does not have a specification, it is directly associated with a classifier (i.e., it is the behavior of the classifier as a whole).
- **context:** BehavioredClassifier [0..1]
The classifier that is the context for the execution of the behavior. If the behavior is owned by a BehavioredClassifier, that classifier is the context. Otherwise, the context is the first BehavioredClassifier reached by following the chain of owner relationships. For example, following this algorithm, the owner of an entry action in a state machine is the classifier that owns the state machine. The features of the context classifier as well as the elements visible to the context classifier are visible to the behavior. (Specializes *RedefinableElement.redefinitionContext*.)
- **ownedParameter:** Parameter
References a list of parameters to the behavior that describes the order and type of arguments that can be given when the behavior is invoked and of the values that will be returned when the behavior completes its execution. (Specializes *Namespace.ownedMember*.)
- **redefinedBehavior:** Behavior
References a behavior that this behavior redefines. A subtype of Behavior may redefine any other subtype of Behavior. If the behavior implements a behavioral feature, it replaces the redefined behavior. If the behavior is a classifier behavior, it extends the redefined behavior.
- **precondition:** Constraint
An optional set of Constraints specifying what must be fulfilled when the behavior is invoked. (Specializes *Namespace.constraint* and *Constraint.context*.)
- **postcondition:** Constraint
An optional set of Constraints specifying what is fulfilled after the execution of the behavior is completed, if its precondition was fulfilled before its invocation. (Specializes *Namespace.constraint* and *Constraint.context*.)

Constraints

- [1] The parameters of the behavior must match the parameters of the implemented behavioral feature.
- [2] The implemented behavioral feature must be a feature (possibly inherited) of the context classifier of the behavior.
- [3] If the implemented behavioral feature has been redefined in the ancestors of the owner of the behavior, then the behavior must realize the latest redefining behavioral feature.
- [4] There may be at most one behavior for a given pairing of classifier (as owner of the behavior) and behavioral feature (as specification of the behavior).

Semantics

The detailed semantics of behavior is determined by its subtypes. The features of the context classifier and elements that are visible to the context classifier are also visible to the behavior, provided that is allowed by the visibility rules.

When a behavior is invoked, its attributes and parameters (if any) are created and appropriately initialized. Upon invocation, the arguments of the original invocation action are made available to the new behavior execution corresponding to its parameters with direction 'in' and 'inout,' if any. When a behavior completes its execution, a value or set of values is returned corresponding to each parameter with direction 'out,' 'inout,' or 'return,' if any. If such a parameter has a default value associated and the behavior does not explicitly generate a value for this parameter, the default value describes the value that will be returned corresponding to this parameter. If the invocation was synchronous, any return values from the behavior execution are returned to the original caller, which is unblocked and allowed to continue execution.

The behavior executes within its context object, independently of and concurrently with any existing behavior executions. The object that is the context of the behavior manages the input pool holding the event occurrences to which a behavior may respond (see 13.3.4, “BehavioredClassifier (from BasicBehaviors, Communications),” on page 419). As an object may have a number of behaviors associated, all these behaviors may access the same input pool. The object ensures that each event occurrence on the input pool is consumed by only one behavior.

When a behavior is instantiated as an object, it is its own context.

Semantic Variation Points

The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication. (See also the discussion on page 409.)

How the parameters of behavioral features or a behavior match the parameters of a behavioral feature is a semantic variation point (see BehavioralFeature on page 418).

Notation

None

Changes from previous UML

This metaclass has been added. It abstracts the commonalities between the various ways that behavior can be implemented in the UML. It allows the various ways of implementing behavior (as expressed by the subtypes of Behavior) to be used interchangeably.

13.3.3 BehavioralFeature (from BasicBehaviors, Communications)

Generalizations

- “BehavioralFeature (from Kernel)” on page 44 (*merge increment*)

Description

A behavioral feature is implemented (realized) by a behavior. A behavioral feature specifies that a classifier will respond to a designated request by invoking its implementing method.

Attributes

Package *BasicBehaviors*

- **isAbstract:** Boolean If *true*, then the behavioral feature does not have an implementation, and one must be supplied by a more specific element.
If *false*, the behavioral feature must have an implementation in the classifier or one must be inherited from a more general element.

Package *Communications*

- **concurrency:** CallConcurrencyKind Specifies the semantics of concurrent calls to the same passive instance (i.e., an instance originating from a class with *isActive* being *false*). Active instances control access to their own behavioral features.

Associations

Package *BasicBehaviors*

- **method:** Behavior A behavioral description that implements the behavioral feature. There may be at most one behavior for a particular pairing of a classifier (as owner of the behavior) and a behavioral feature (as specification of the behavior).

Package *Communications*

- **raisedException:** Classifier The signals that the behavioral feature raises as exceptions. (Specializes *BehavioralFeature.raisedException*.)

Constraints

No additional constraints

Semantics

The invocation of a method is caused by receiving a request invoking the behavioral feature specifying that behavior. The details of invoking the behavioral feature are defined by the subclasses of *BehavioralFeature*.

Semantic Variation Points

How the parameters of behavioral features or a behavior match the parameters of a behavioral feature is a semantic variation point. Different languages and methods rely on exact match (i.e., the type of the parameters must be the same), co-variant match (the type of a parameter of the behavior may be a subtype of the type of the parameter of the behavioral feature), contra-variant match (the type of a parameter of the behavior may be a supertype of the type of the parameter of the behavioral feature), or a combination thereof.

Changes from previous UML

The metaattributes *isLeaf* and *isRoot* have been replaced by properties inherited from *RedefinableElement*.

13.3.4 BehavoredClassifier (from BasicBehaviors, Communications)

Generalizations

- “Class (from Kernel)” on page 45

Description

A classifier can have behavior specifications defined in its namespace. One of these may specify the behavior of the classifier itself.

Attributes

No additional attributes

Associations

- ownedBehavior: Behavior References behavior specifications owned by a classifier. (Specializes *Namespace.ownedMember*.)
- classifierBehavior: Behavior [0..1] A behavior specification that specifies the behavior of the classifier itself. (Specializes *BehavioredClassifier.ownedBehavior*.)

Package Communications

- ownedTrigger : Trigger [0..*] References Trigger descriptions owned by a Classifier (Specializes *Namespace.ownedMember*.)

Constraints

If a behavior is classifier behavior, it does not have a specification.

Semantics

The behavior specifications owned by a classifier are defined in the context of the classifier. Consequently, the behavior specifications may reference features of the classifier. Any invoked behavior may, in turn, invoke other behaviors visible to its context classifier. When an instance of a behaviored classifier is created, its classifier behavior is invoked.

When an event occurrence is recognized by an object that is an instance of a behaviored classifier, it may have an immediate effect or the occurrence may be saved for later *triggered* effect. An immediate effect is manifested by the invocation of a behavior as determined by the event (the type of the occurrence). A triggered effect is manifested by the storage of the occurrence in the input event pool of the object and the later consumption of the occurrence by the execution of an ongoing behavior that reaches a point in its execution at which a trigger matches the event (type) of the occurrence in the pool. At this point, a behavior may be invoked as determined by the event.

When an executing behavior owned by an object comes to a point where it needs a trigger to continue its execution, the input pool is examined for an event that satisfies the outstanding trigger or triggers. If an event satisfies one of the triggers, the event is removed from the input pool and the behavior continues its execution, as specified. Any data associated with the event are made available to the triggered behavior.

Semantic Variation Points

It is a semantic variation whether one or more behaviors are triggered when an event satisfies multiple outstanding triggers.

If an event in the pool satisfies no triggers at a wait point, it is a semantic variation point what to do with it.

The ordering of the events in the input pool is a semantic variation.

Notation

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

Changes from previous UML

In UML 1.4, there was no separate metaclass for classifiers with behavior.

13.3.5 CallConcurrencyKind (from Communications)

Generalizations

None

Description

CallConcurrencyKind is an enumeration with the following literals:

- sequential - No concurrency management mechanism is associated with the operation and, therefore, concurrency conflicts may occur. Instances that invoke a behavioral feature need to coordinate so that only one invocation to a target on any behavioral feature occurs at once.
- guarded - Multiple invocations of a behavioral feature may occur simultaneously to one instance, but only one is allowed to commence. The others are blocked until the performance of the first behavioral feature is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks.
- concurrent - Multiple invocations of a behavioral feature may occur simultaneously to one instance and all of them may proceed concurrently.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See Description section above.

Notation

None

Changes from previous UML

None

13.3.6 CallEvent (from Communications)

A CallEvent models the receipt by an object of a message invoking a call of an operation.

Generalizations

- “MessageEvent (from Communications)” on page 431

Description

A call event represents the *reception* of a request to invoke a specific operation. A call event is distinct from the call action that caused it. A call event may cause the invocation of a behavior that is the method of the operation referenced by the call request, if that operation is owned or inherited by the classifier that specified the receiver object.

Attributes

No additional attributes

Associations

- operation: Operation [1] Designates the operation whose invocation raised the call event.

Constraints

No additional constraints

Semantics

A call event represents the reception of a request to invoke a specific operation on an object. The call event may result in the execution of the behavior that implements the called operation. A call event may, in addition, cause other responses, such as a state machine transition, as specified in the classifier behavior of the classifier that specified the receiver object. In that case, the additional behavior is invoked after the completion of the operation referenced by the call trigger.

A call event makes available any argument values carried by the received call request to all behaviors caused by this event (such as transition actions or entry actions).

Notation

Call events are denoted by a list of names of the triggering operations, followed by an assignment specification:

$$\langle \text{call-event} \rangle ::= \langle \text{name} \rangle [(' [\langle \text{assignment-specification} \rangle] ')]$$
$$\langle \text{assignment-specification} \rangle ::= \langle \text{attr-name} \rangle [, ' \langle \text{attr-name} \rangle] ^*$$

where:

- $\langle \text{attr-name} \rangle$ is an implicit assignment of the corresponding parameter of the operation to an attribute (with this name) of the context object owning the triggered behavior.

$\langle \text{assignment-specification} \rangle$ is optional and may be omitted even if the operation has parameters.

13.3.7 ChangeEvent (from Communications)

A change event models a change in the system configuration that makes a condition true.

Generalizations

- “Event (from Communications)” on page 428

Description

A change event occurs when a Boolean-valued expression becomes true. For example, as a result of a change in the value held in a slot corresponding to an attribute, or a change in the value referenced by a link corresponding to an association. A change event is raised implicitly and is *not* the result of an explicit action.

Attributes

No additional attributes

Associations

- **changeExpression:** Expression [1] A Boolean-valued expression that will result in a change event whenever its value changes from *false* to *true*.

Constraints

No additional constraints

Semantics

Each time the value of the change expression changes from false to true, a change event is generated.

Semantic Variation Points

It is a semantic variation when the change expression is evaluated. For example, the change expression may be continuously evaluated until it becomes *true*. It is further a semantic variation whether a change event remains until it is consumed, even if the change expression changes to *false* after a change event.

Notation

A change event is denoted in a trigger by a Boolean expression.

<change-event> ::= 'when' <expression>

Changes from previous UML

This metaclass replaces change event.

13.3.8 Class (from Communications)

Generalizations

- “BehavioredClassifier (from BasicBehaviors, Communications)” on page 419
- “Class (from Kernel)” on page 45 (*merge increment*)

Description

A class may be designated as active (i.e., each of its instances having its own thread of control) or passive (i.e., each of its instances executing within the context of some other object).

A class may also specify which signals the instances of this class handle.

Attributes

- **isActive:** Boolean Determines whether an object specified by this class is active or not. If *true*, then the owning class is referred to as an *active class*. If *false*, then such a class is referred to as a *passive class*.

Associations

- ownedReception: Reception Receptions that objects of this class are willing to accept. (Specializes *Namespace.ownedMember* and *Classifier.feature*.)

Semantics

An active object is an object that, as a direct consequence of its creation, commences to execute its classifier behavior, and does not cease until either the complete behavior is executed or the object is terminated by some external object. (This is sometimes referred to as “the object having its own thread of control.”) The points at which an active object responds to communications from other objects is determined solely by the behavior of the active object and not by the invoking object. If the classifier behavior of an active object completes, the object is terminated.

Notation

See presentation options below.

Presentation options

A class with the property *isActive* = *true* can be shown by a class box with an additional vertical bar on either side, as depicted in Figure 13.14.



Figure 13.14 - Active class

13.3.9 Duration (from SimpleTime)

Generalizations

- “ValueSpecification (from Kernel)” on page 132

Description

A duration defines a value specification that specifies the temporal distance between two time expressions that specify time instants.

Attributes

- firstTime:Boolean [0..2]
If the duration is between times of two NamedElements, there are two Boolean attributes, one for the start of the duration and one for the end of the duration. For each of these it holds that *firstTime* is *true* if the time information is associated with the first point in time of the NamedElement referenced by *event*, and *false* if it represents the last point in time of the NamedElement. If there is only one NamedElement referenced by *event*, then this attribute is irrelevant. The default value is *true*.

Associations

- event: NamedElement [0..2]
Refers to the specification(s) that describes the starting TimeExpression and the ending TimeExpression of the Duration.

If only one NamedElement is referenced, the duration is from the first point in time of that NamedElement until the last point in time of that NamedElement.

Constraints

No additional constraints

Semantics

A Duration defines a ValueSpecification that denotes some duration in time. The duration is given by the difference in time between a starting point in time and an ending point in time.

If the ending point in time precedes the starting point in time, the duration will still be positive assuming the starting point and ending points to swap.

Notation

A Duration is a value of relative time given in an implementation specific textual format. Often a Duration is a non-negative integer expression representing the number of “time ticks,” which may elapse during this duration.

Changes from previous UML

This metaclass has been added.

13.3.10 DurationConstraint (from SimpleTime)

Generalizations

- “IntervalConstraint (from SimpleTime)” on page 430

Description

A DurationConstraint defines a Constraint that refers to a DurationInterval.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics of a DurationConstraint is inherited from Constraints.

Notation

A DurationConstraint is shown as some graphical association between a DurationInterval and the constructs that it constrains. The notation is specific to the diagram type.

Examples

See example in Figure 13.15 where the TimeConstraint is associated with the duration of a Message and the duration between two EventOccurrences.

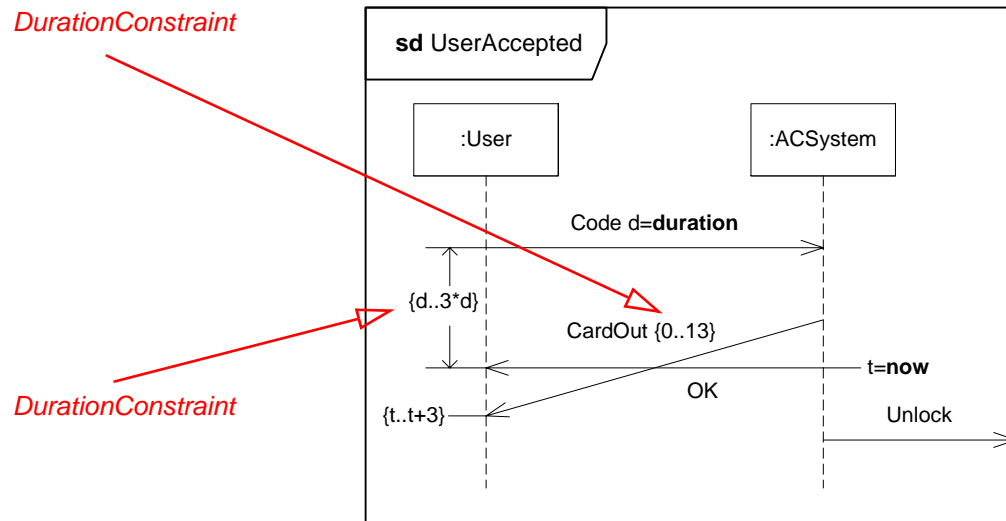


Figure 13.15 - DurationConstraint and other time-related concepts

Changes from previous UML

This metaclass has been added.

13.3.11 DurationInterval (from SimpleTime)

Generalizations

- “Interval (from SimpleTime)” on page 430

Description

A DurationInterval defines the range between two Durations.

Attributes

No additional attributes

Associations

- min: Duration [1] Refers to the Duration denoting the minimum value of the range.
- max: Duration [1] Refers to the Duration denoting the maximum value of the range.

Constraints

No additional constraints

Semantics

None

Notation

A DurationInterval is shown using the notation of Interval where each value specification element is a DurationExpression.

13.3.12 DurationObservationAction (from SimpleTime)

Generalizations

- “WriteStructuralFeatureAction (from IntermediateActions)” on page 282

Description

A DurationObservationAction defines an action that observes duration in time and writes this value to a structural feature.

Attributes

No additional attributes

Associations

- duration: Duration[1] represents the measured Duration.

Constraints

No additional constraints

Semantics

A DurationObservationAction is an action that, when executed, measures an identified duration in the context in which it is executing and writes the obtained value to the given structural feature.

Notation

A Duration is depicted by text in the expression language used to denote a duration value. It may be possible that a duration contains arithmetic operators. A duration observation occurs when a duration is assigned to a structural feature. The duration observation is associated with two NamedElements with lines.

<durationobservation> ::= <write-once-attribute> '= duration'

Examples

See example in Figure 13.16 where the duration observation records the duration of a message (i.e., the time between the sending and the reception of that message).

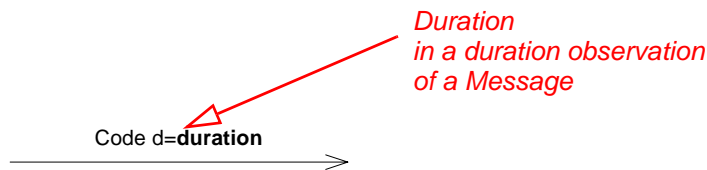


Figure 13.16 - Duration observation

Changes from previous UML

This metaclass has been added.

13.3.13 Event (from Communications)

Generalizations

- “PackageableElement (from Kernel)” on page 105

Description

An event is the specification of some occurrence that may potentially trigger effects by an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An event is the specification of some occurrence that may potentially trigger effects by an object. This is an abstract metaclass.

Notation

None

Changes from previous UML

None

13.3.14 FunctionBehavior (from BasicBehaviors)

Generalizations

- “OpaqueBehavior (from BasicBehaviors)” on page 432

Description

A function behavior is an opaque behavior that does not access or modify any objects or other external data.

Attributes

None

Associations

None

Constraints

- [1] A function behavior has at least one output parameter.
- [2] The types of parameters are all data types, which may not nest anything but other datatypes.

Semantics

Primitive functions transform a set of input values to a set of output values by invoking a function. They represent functions from a set of input values to a set of output values. The execution of a primitive function depends only on the input values and has no other effect than to compute output values. A primitive function does not read or write structural feature or link values, nor otherwise interact with object memory or other objects. Its behavior is completely self-contained. Specific primitive functions are not defined in the UML, but would be defined in domain-specific extensions. Typical primitive functions would include arithmetic, Boolean, and string functions.

During the execution of the function, no communication or interaction with the rest of the system is possible. The amount of time to compute the results is undefined. FunctionBehavior may raise exceptions for certain input values, in which case the computation is abandoned.

Notation

None

Examples

Mathematical functions are examples of function behaviors.

Rationale

FunctionBehavior is introduced to model external functions that only take inputs and produce outputs and have no effect on the specified system.

13.3.15 Interface (from Communications)

Generalizations

- “Interface (from Interfaces)” on page 82 (*merge increment*)

Description

Adds the capability for interfaces to include receptions (in addition to operations).

Associations

- ownedReception: Reception — Receptions that objects providing this interface are willing to accept. (Subsets *Namespace.ownedMember* and *Classifier.feature*)

13.3.16 Interval (from SimpleTime)

Generalizations

- “ValueSpecification (from Kernel)” on page 132

Description

An Interval defines the range between two value specifications.

Attributes

No additional attributes

Associations

- min: ValueSpecification[1] Refers to the ValueSpecification denoting the minimum value of the range.
- max: ValueSpecification[1] Refers to the ValueSpecification denoting the maximum value of the range.

Constraints

No additional constraints

Semantics

The semantics of an Interval is always related to Constraints in which it takes part.

Notation

An Interval is denoted textually by two ValueSpecifications separated by “..”:

<interval> ::= <min-value> ‘..’ <max-value>

Changes from previous UML

This metaclass has been added.

13.3.17 IntervalConstraint (from SimpleTime)

Generalizations

- “Constraint (from Kernel)” on page 54

Description

An IntervalConstraint defines a Constraint that refers to an Interval.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics of an IntervalConstraint is inherited from Constraint. All traces where the constraints are violated are negative traces (i.e., if they occur in practice the system has failed).

Notation

An IntervalConstraint is shown as a graphical association between an Interval and the constructs that this Interval constrains. The concrete form is given in its subclasses.

Changes from previous UML

This metaclass has been added.

13.3.18 MessageEvent (from Communications)**Generalizations**

- “Event (from Communications)” on page 428

Description

A message event specifies the receipt by an object of either a call or a signal. MessageEvent is an abstract metaclass.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

The metaclass has been added.

13.3.19 OpaqueBehavior (from BasicBehaviors)

Generalizations

- “Behavior (from BasicBehaviors)” on page 416

Description

A behavior with implementation-specific semantics.

Attributes

- body : String [1..*] Specifies the behavior in one or more languages.
- language : String [*] Languages the body strings use in the same order as the body strings.

Associations

None

Constraints

None

Semantics

The semantics of the behavior is determined by the implementation.

Notation

None

Rationale

OpaqueBehavior is introduced for implementation-specific behavior or for use as a place-holder before one of the other behaviors is chosen.

13.3.20 OpaqueExpression (from BasicBehaviors)

Generalizations

- “OpaqueExpression (from Kernel)” on page 97 (*merge increment*)

Description

Provides a mechanism for precisely defining the behavior of an opaque expression. An opaque expression is defined by a behavior restricted to return one result.

Attributes

No additional attributes

Associations

- behavior: Behavior [0..1] Specifies the behavior of the opaque expression.

- **result: Parameter [0..1]** Restricts an opaque expression to return exactly one return result. When the invocation of the opaque expression completes, a single set of values is returned to its owner. This association is derived from the single return result parameter of the associated behavior.

Constraints

- [1] The *behavior* must not have formal parameters.
- [2] The *behavior* must have exactly one return result parameter.

Semantics

An opaque expression is invoked by the execution of its owning element. An opaque expression does not have formal parameters and thus cannot be passed data upon invocation. It accesses its input data through elements of its behavioral description. Upon completion of its execution, a single value or a single set of values is returned to its owner.

13.3.21 Operation (from Communications)

Generalizations

- “Operation (from Kernel, Interfaces)” on page 99 (*merge increment*)

Description

An operation may invoke both the execution of method behaviors as well as other behavioral responses.

Semantics

If an operation is not mentioned in a trigger of a behavior owned or inherited by the behavior classifier owning the operation, then upon occurrence of a call event (representing the receipt of a request for the invocation of this operation) a resolution process is performed that determines the method behavior to be invoked, based on the operation and the data values corresponding to the parameters of the operation transmitted by the request. Otherwise, the call event is placed into the input pool of the object (see BehaviorClassifier on page 419). If a behavior is triggered by this event, it begins with performing the resolution process and invoking the so determined method. Then the behavior continues its execution as specified.

Operations specify immediate or triggered effects (see “BehaviorClassifier” on page 419).

Semantic Variation Points

Resolution specifies how a particular behavior is identified to be executed in response to the invocation of an operation, using mechanisms such as inheritance. The mechanism by which the behavior to be invoked is determined from an operation and the transmitted argument data is a semantic variation point. In general, this mechanism may be complicated to include languages with features such as before-after methods, delegation, etc. In some of these variations, multiple behaviors may be executed as a result of a single call. The following defines a simple object-oriented process for this semantic variation point.

- **Object-oriented resolution**
When a call request is received, the class of the target object is examined for an owned operation with matching parameters (see “BehavioralFeature” on page 418). If such operation is found, the behavior associated as *method* is the result of the resolution. Otherwise the parent classifier is examined for a matching operation, and so on up the generalization hierarchy until a method is found or the root of the hierarchy is reached. If a class has multiple parents, all of them are examined for a method. If a method is found in exactly one ancestor class, then that method is the result of the

resolution. If a method is found in more than one ancestor class along different paths, then the model is ill formed under this semantic variation.

If no method by the resolution process, then it is a semantic variation point what is to happen.

13.3.22 Reception (from Communications)

Generalizations

- “BehavioralFeature (from Kernel)” on page 44

Description

A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal. A reception designates a signal and specifies the expected behavioral response. The details of handling a signal are specified by the behavior associated with the reception or the classifier itself.

Attributes

No additional attributes

Associations

- signal: Signal [0..1] The signal that this reception handles.

Constraints

- [1] A Reception cannot be a query.
not self.isQuery
- [2] A passive class cannot have receptions.

Semantics

The receipt of a signal instance by the instance of the classifier owning a matching reception will cause the asynchronous invocation of the behavior specified as the method of the reception. A reception matches a signal if the received signal is a subtype of the signal referenced by the reception. The details of how the behavior responds to the received signal depends on the kind of behavior associated with the reception. (For example, if the reception is implemented by a state machine, the signal event will trigger a transition and subsequent effects as specified by that state machine.)

Receptions specify triggered effects (see “BehavioredClassifier” on page 419).

Notation

Receptions are shown using the same notation as for operations with the keyword «signal», as shown in Figure 13.17.

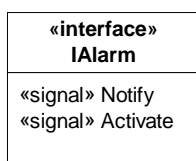


Figure 13.17 - Showing signal receptions in classifiers

Changes from previous UML

None

13.3.23 Signal (from Communications)

Generalizations

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

Description

A signal is a specification of type of send request instances communicated between objects. The receiving object handles the signal instance as specified by its receptions. The data carried by a send request and passed to it by the occurrence of the send invocation event that caused the request is represented as attributes of the signal instance. A signal is defined independently of the classifiers handling the signal.

Attributes

No additional attributes

Associations

- ownedAttribute: Property [*] The attributes owned by the signal. The association is ordered and subsets *Classifier.attribute* and *Namespace.ownedMember*.

Constraints

No additional constraints

Semantics

A signal triggers a reaction in the receiver in an asynchronous way and without a reply. The sender of a signal will not block waiting for a reply but continue execution immediately. By declaring a reception associated to a given signal, a classifier specifies that its instances will be able to receive that signal, or a subtype thereof, and will respond to it with the designated behavior.

Notation

A signal is depicted by a classifier symbol with the keyword «signal».

Changes from previous UML

None

13.3.24 SignalEvent (from Communications)

A signal event represents the *receipt* of an asynchronous signal instance. A signal event may, for example, cause a state machine to trigger a transition.

Generalizations

- “MessageEvent (from Communications)” on page 431

Description

A signal event represents the *receipt* of an asynchronous signal. A signal event may cause a response, such as a state machine transition as specified in the classifier behavior of the classifier that specified the receiver object, if the signal referenced by the send request is mentioned in a reception owned or inherited by the classifier that specified the receiver object.

Attributes

- signal: Signal [1] The specific signal that is associated with this event.

Associations

No additional associations

Constraints

No additional constraints

Semantics

A signal event occurs when a signal message, originally caused by a send action executed by some object, is received by another (possibly the same) object. A signal event may result in the execution of the behavior that implements the reception matching the received signal.

A signal event makes available any argument values carried by the received send request to all behaviors caused by this event (such as transition actions or entry actions).

Semantic Variation Points

The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication. (See also the discussion on page 409.)

Notation

Signal events are denoted by a list of names of the triggering signals, followed by an assignment specification:

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

where:

- $\langle \text{attr-name} \rangle$ is an implicit assignment of the corresponding parameter of the signal to an attribute (with this name) of the context object owning the triggered behavior.
- $\langle \text{assignment-specification} \rangle$ is optional and may be omitted even if the signal has parameters.

Changes from previous UML

This metaclass replaces SignalEvent.

13.3.25 TimeConstraint (from SimpleTime)

Generalizations

- “IntervalConstraint (from SimpleTime)” on page 430

Description

A TimeConstraint defines a Constraint that refers to a TimeInterval.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics of a TimeConstraint is inherited from Constraints. All traces where the constraints are violated are negative traces (i.e., if they occur in practice, the system has failed).

Notation

A TimeConstraint is shown as graphical association between a TimeInterval and the construct that it constrains. Typically this graphical association is a small line (e.g., between an EventOccurrence and a TimeInterval).

Examples

See example in Figure 13.18 where the TimeConstraint is associated with the reception of a Message.

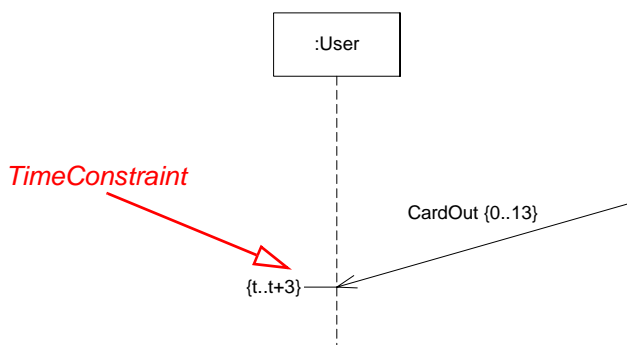


Figure 13.18 - TimeConstraint

Changes from previous UML

This metaclass has been added.

13.3.26 TimeEvent (from Communications, SimpleTime)

A TimeEvent specifies a point in time. At the specified time, the event occurs.

Generalizations

- “Event (from Communications)” on page 428

Description

A time event specifies a point in time by an expression. The expression might be absolute or might be relative to some other point in time.

Attributes

- `isRelative`: Boolean Specifies whether it is relative or absolute time.

Associations

- `when`: TimeExpression [1] Specifies the corresponding time deadline.

Constraints

No additional constraints

Semantics

A time event specifies an instant in time by an expression. The expression might be absolute or it might be relative to some other point in time. Relative time events must always be used in the context of a trigger and the starting point is the time at which the trigger becomes active.

Semantic Variation Points

There may be a variable delay between the time of reception and the time of dispatching of the TimeEvent (e.g., due to queueing delays).

Notation

A relative time trigger is specified with the keyword ‘after’ followed by an expression that evaluates to a time value, such as “after (5 seconds).” An absolute time trigger is specified with the keyword ‘at’ followed by an expression that evaluates to a time value, such as “Jan. 1, 2000, Noon.”

```
<time-event> ::= <relative-time-event> / <absolute-time-event>
<relative-time-event> ::= ‘after’ <expression>
<absolute-time-event> ::= ‘at’ <expression>
```

Changes from previous UML

The attribute *isRelative* has been added for clarity.

13.3.27 TimeExpression (from SimpleTime)

Generalizations

- “ValueSpecification (from Kernel)” on page 132

Description

A TimeExpression defines a value specification that represents a time value.

Attributes

- firstTime:Boolean *True* if the TimeExpression describes the first point in time of the NamedElement referenced by *event*, in cases where the NamedElement describes something that extends in time.
False if the TimeExpression describes the last point in time for the referenced NamedElement.

Associations

- event: NamedElement [0..1] Refers to the specification of the event occurrence that the TimeExpression describes.

Constraints

No additional constraints

Semantics

A TimeExpression denotes a time value.

Notation

A TimeExpression is a value of absolute time given in an implementation specific textual format. Often a TimeExpression is a non-negative integer expression representing the number of “time ticks” after some given starting point.

Changes from previous UML

This metaclass has been added.

13.3.28 TimeInterval (from SimpleTime)

Generalizations

- “Interval (from SimpleTime)” on page 430.

Description

A TimeInterval defines the range between two TimeExpressions.

Attributes

No additional attributes

Associations

- min: TimeExpression [1] Refers to the TimeExpression denoting the minimum value of the range.
- max: TimeExpression [1] Refers to the TimeExpression denoting the maximum value of the range.

Constraints

No additional constraints

Semantics

None

Notation

A TimeInterval is shown with the notation of Interval where each value specification element is a TimeExpression.

Changes from previous UML

This metaclass has been added.

13.3.29 TimeObservationAction (from SimpleTime)

Generalizations

- “WriteStructuralFeatureAction (from IntermediateActions)” on page 282

Description

A TimeObservationAction defines an action that observes the current point in time and writes this value to a structural feature.

Attributes

No additional attributes

Associations

- **now**: TimeExpression [1] Represents the current point in time and the value that is observed given by the keyword **now**.

Constraints

No additional constraints

Semantics

A TimeObservationAction is an action that, when executed, obtains the current value of time in the context in which it is executing and writes this value to the given structural feature.

Notation

A TimeExpression is depicted by text in the expression language used to denote a time value. It may be possible that a time value contains arithmetic operators. The time expression is associated with a NamedElement with a line. A time observation action assigns a time expression to a structural feature.

<timeobservation> ::= <write-once-attribute> ‘= now’

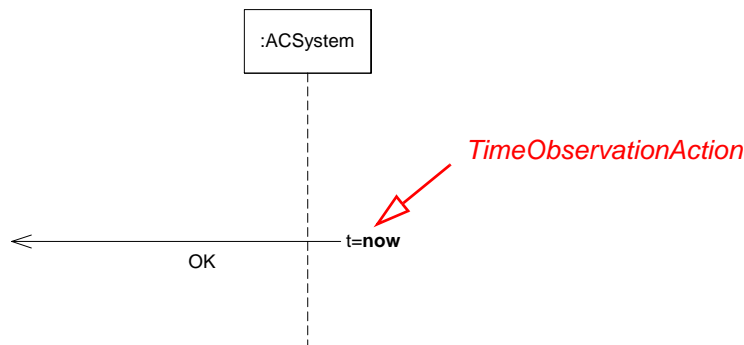


Figure 13.19 - Time observation

Changes from previous UML

This metaclass has been added.

13.3.30 Trigger (from Communications)

A trigger relates an event to a behavior that may affect an instance of the classifier.

Generalizations

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

Description

A trigger specifies an event that may cause the execution of an associated behavior. An event is often ultimately caused by the execution of an action, but need not be.

Attributes

No additional attributes

Associations

- port: Port [*] Optionally specifies the ports at which a communication that caused an event may have arrived.
- event : Event [1] The event that causes the trigger.

Constraints

No additional constraints

Semantics

Events may cause execution of behavior (e.g., the execution of the effect activity of a transition in a state machine). A trigger specifies the event that may trigger a behavior execution as well as any constraints on the event to filter out events not of interest.

Events are often generated as a result of some action either within the system or in the environment surrounding the system. Upon their occurrence, events are placed into the input pool of the object where they occurred (see BehavedClassifier on page 419). An event is dispatched when it is taken from the input pool and either directly causes

the occurrence of a behavior or are delivered to the classifier behavior of the receiving object for processing. At this point, the event is considered consumed and referred to as the current event. A consumed event is no longer available for processing.

Semantic Variation Points

No assumptions are made about the time intervals between event occurrence, event dispatching, and consumption. This leaves open the possibility of different semantic variations such as zero-time semantics.

It is a semantic variation whether an event is discarded if there is no appropriate trigger defined for them.

Notation

A trigger is used to define an unnamed event. The details of the syntax for the event are defined by the different subclasses of Event:

<trigger> ::= <call-event> / <signal-event> / <any-receive-event> / <time-event> / <change-event>

Changes from previous UML

The corresponding metaclass in 1.x was Event. In 1.x, events were specified with Parameters. Instead, the data that may be communicated by an event is accessed via the properties of the specification element defining the event.

14 Interactions

14.1 Overview

Interactions are used in a number of different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases.

The Interaction package describes the concepts needed to express Interactions, depending on their purpose. An interaction can be displayed in several different types of diagrams: Sequence Diagrams, Interaction Overview Diagrams, and Communication Diagrams. Optional diagram types such as Timing Diagrams and Interaction Tables come in addition. Each type of diagram provides slightly different capabilities that makes it more appropriate for certain situations.

Interactions are a common mechanism for describing systems that can be understood and produced, at varying levels of detail, by both professionals of computer systems design, as well as potential end users and stakeholders of (future) systems.

Typically when interactions are produced by designers or by running systems, the case is that the interactions do not tell the complete story. There are normally other legal and possible traces that are not contained within the described interactions. Some projects do, however, request that all possible traces of a system shall be documented through interactions in the form of (e.g., sequence diagrams or similar notations).

The most visible aspects of an Interaction are the messages between the lifelines. The sequence of the messages is considered important for the understanding of the situation. The data that the messages convey and the lifelines store may also be very important, but the Interactions do not focus on the manipulation of data even though data can be used to decorate the diagrams.

In this chapter we use the term *trace* to mean “sequence of event occurrences,” which corresponds well with common use in the area of trace-semantics, which is a preferred way to describe the semantics of Interactions. We may denote this by `<eventoccurrence1, eventoccurrence2, ...,eventoccurrence-n>`. We are aware that other parts of the UML language definition of the term “trace” is used also for other purposes.

By *interleaving* we mean the merging of two or more traces such that the events from different traces may come in any order in the resulting trace, while events within the same trace retain their order. Interleaving semantics is different from a semantics where it is perceived that two events may occur at exactly the same time. To explain Interactions we apply an Interleaving Semantics.

14.2 Abstract syntax

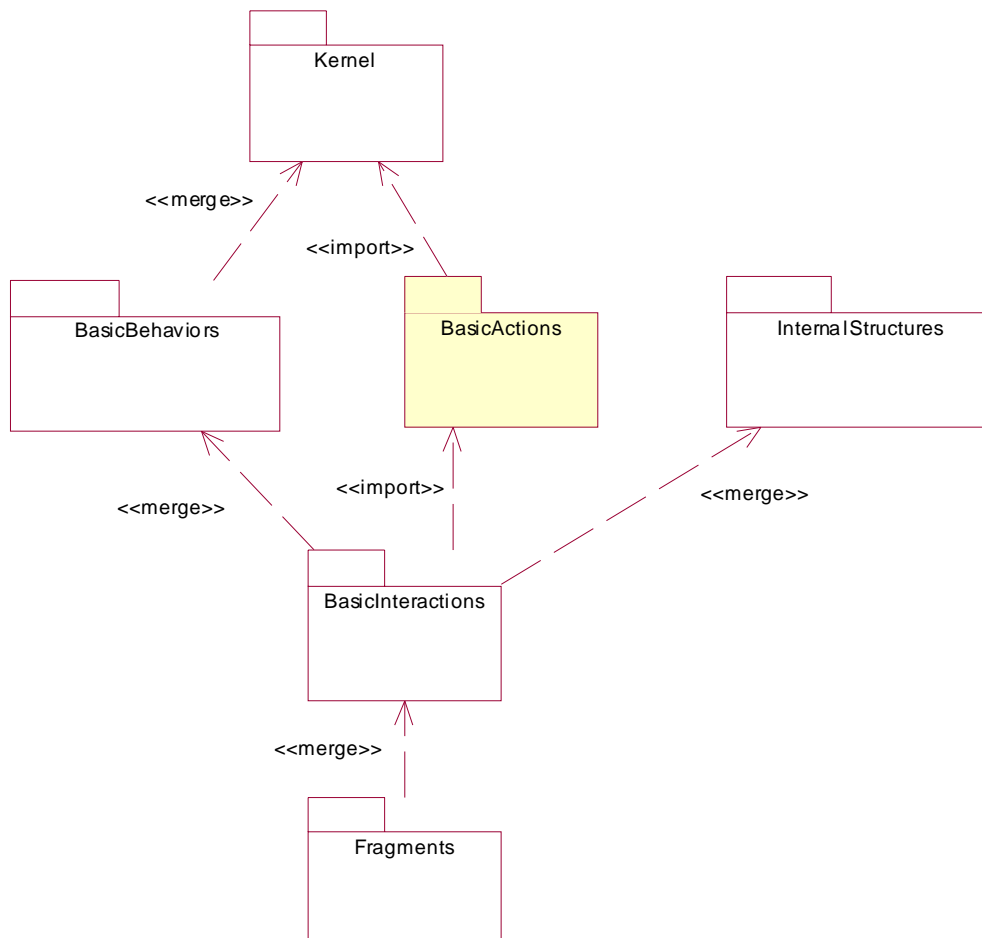


Figure 14.1 - Dependencies of the Interactions packages

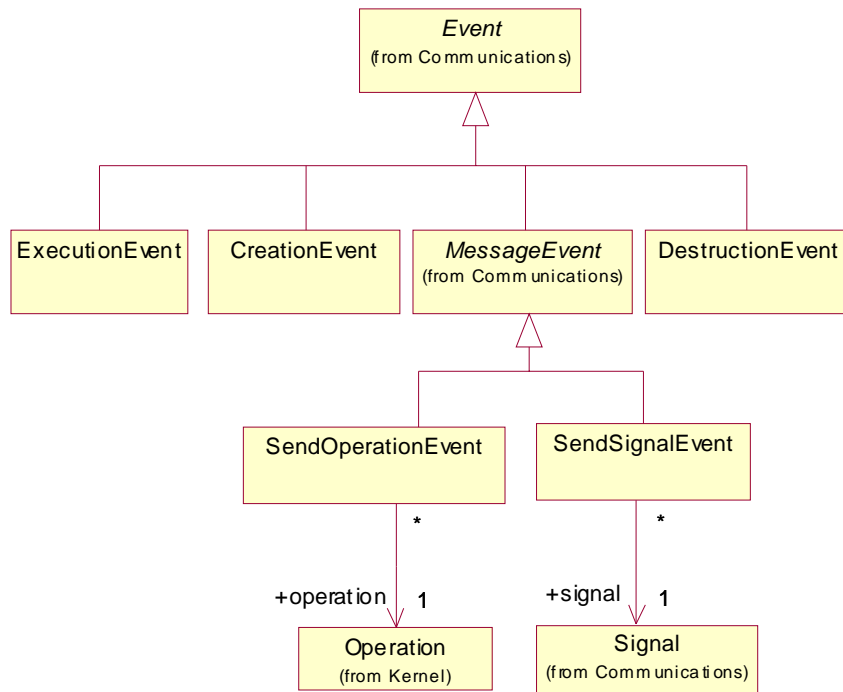


Figure 14.2 Events

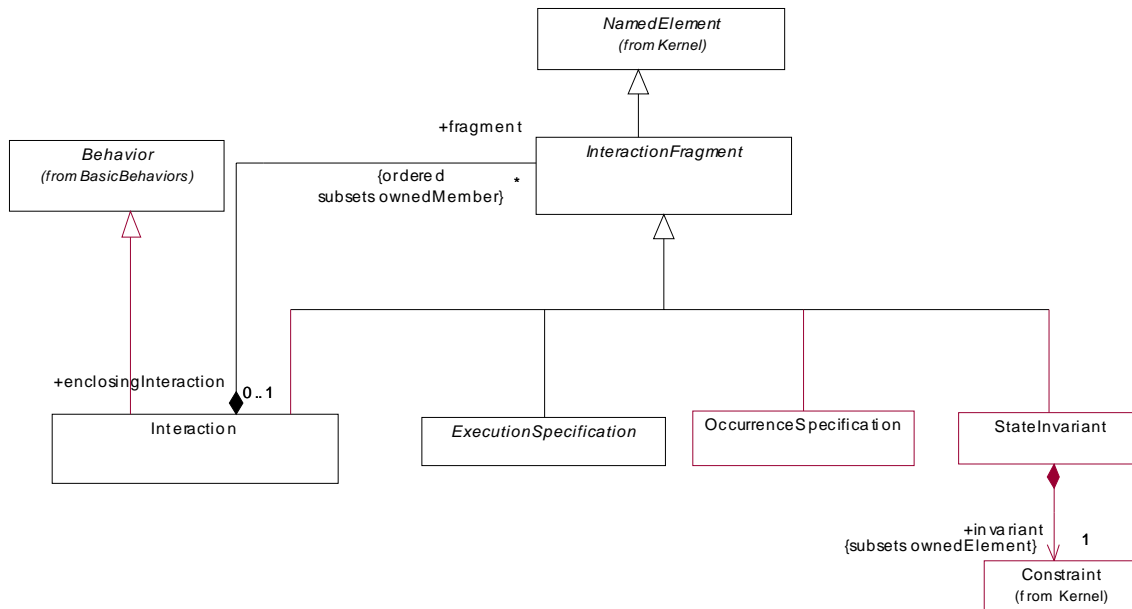


Figure 14.3 - Interactions

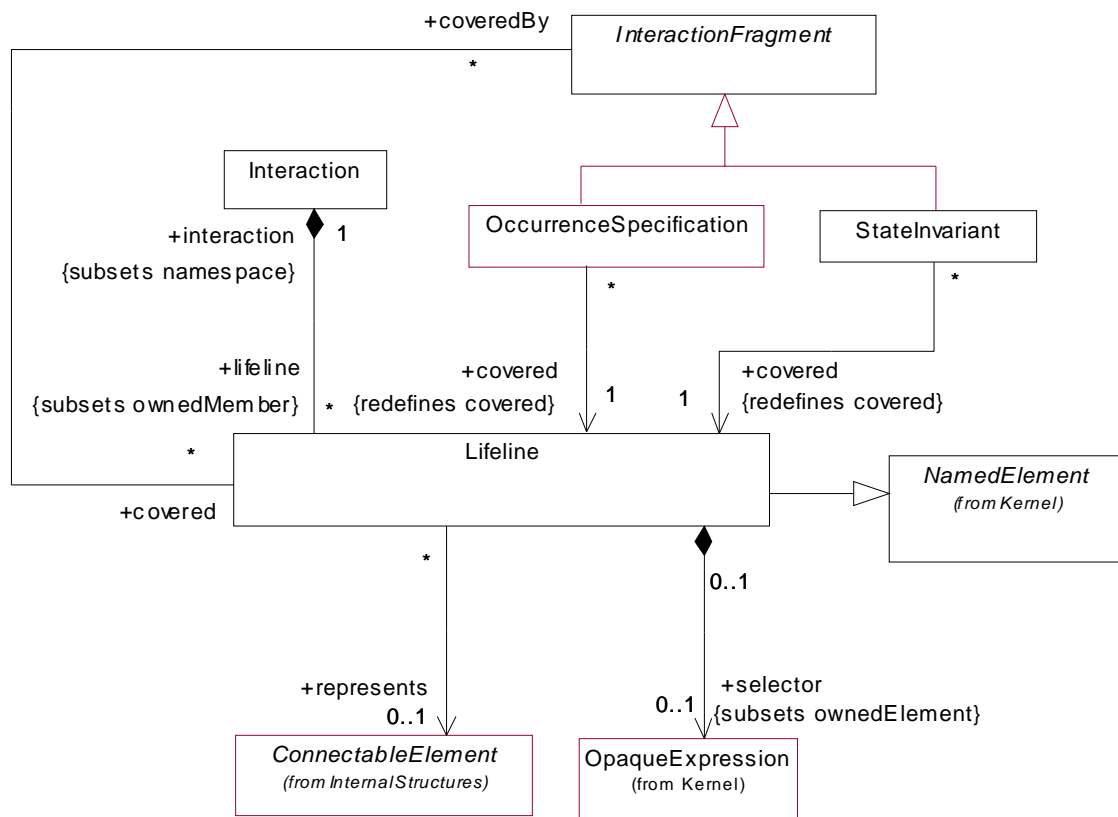


Figure 14.4 - Lifelines

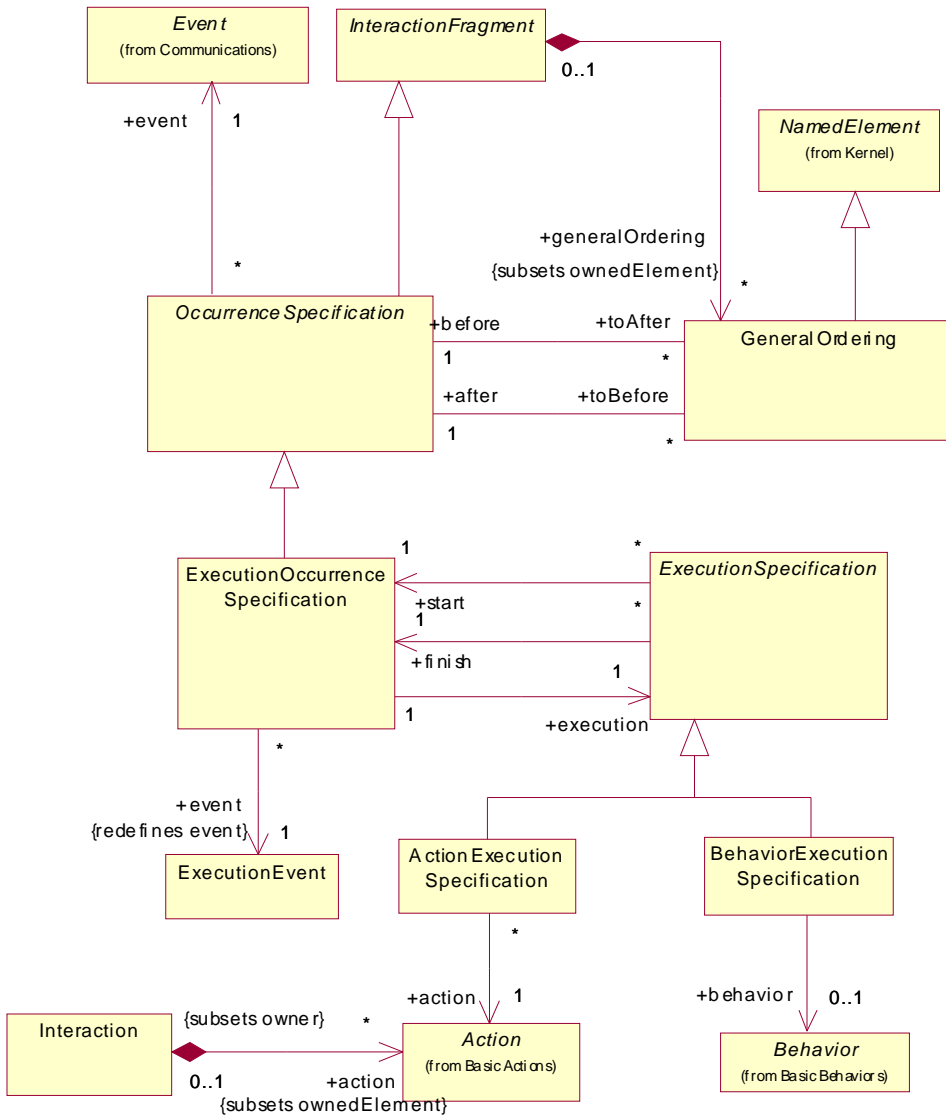


Figure 14.6 Occurrences

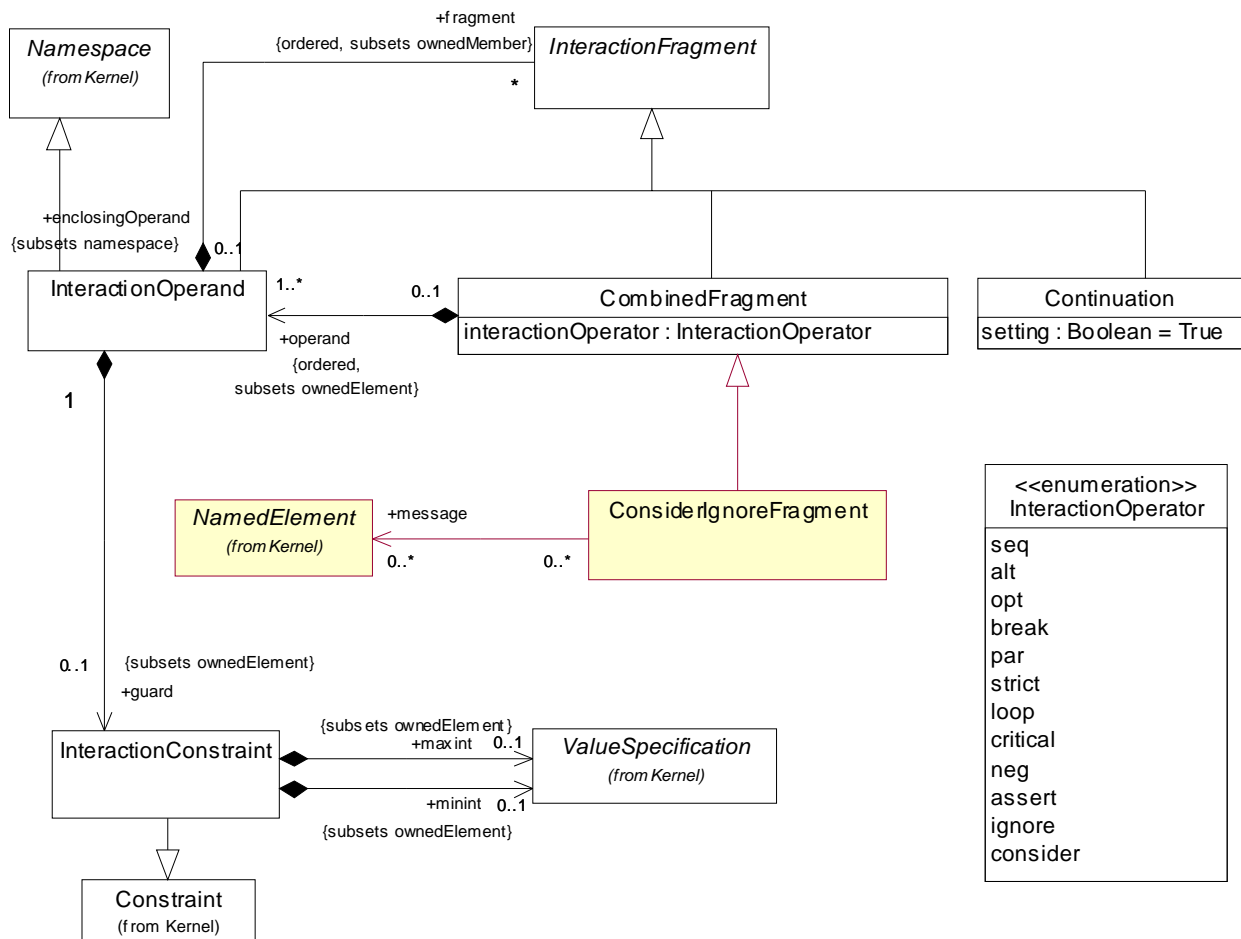


Figure 14.7 - CombinedFragments

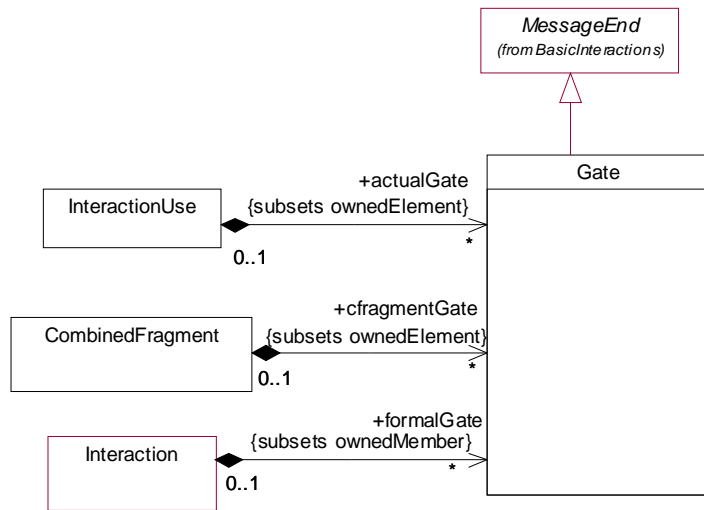


Figure 14.8 - Gates

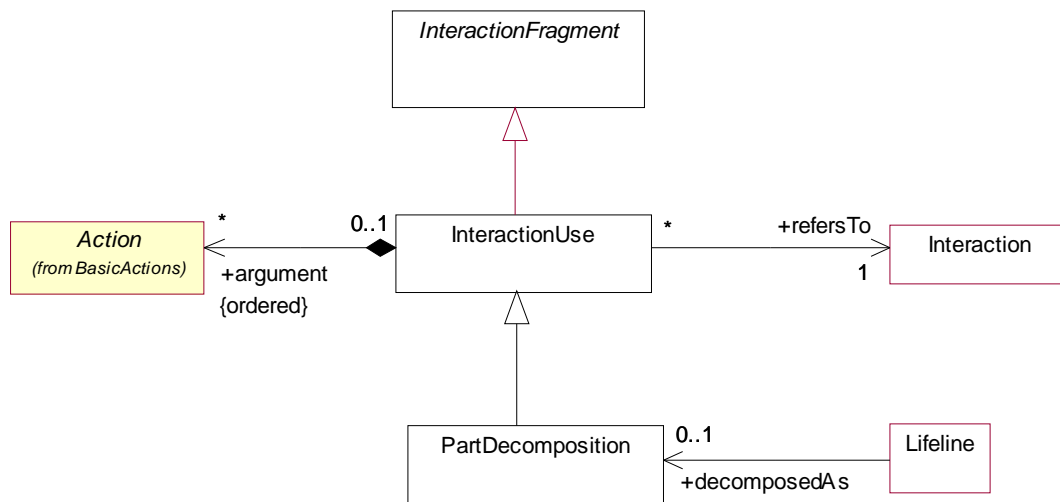


Figure 14.9 - InteractionUses

14.3 Class Descriptions

14.3.1 ActionExecutionSpecification (from BasicInteractions)

Generalizations

- “ExecutionSpecification (from BasicInteractions)” on page 464.

Description

ActionExecutionSpecification is a kind of ExecutionSpecification representing the execution of an action.

Attributes

No additional attributes

Associations

- action : Action [1] Action whose execution is occurring.

Constraints

[1] The Action referenced by the ActionExecutionOccurrence, if any, must be owned by the Interaction owning the ActionExecutionOccurrence.

Semantics

See “ExecutionSpecification (from BasicInteractions)” on page 464.

Notation

See “ExecutionSpecification (from BasicInteractions)” on page 464.

Rationale

ActionExecutionSpecification is introduced to support interactions specifying messages that result from actions, which may be actions owned by other behaviors.

14.3.2 BehaviorExecutionSpecification (from BasicInteractions)

Generalizations

- “ExecutionSpecification (from BasicInteractions)” on page 464

Description

BehaviorExecutionSpecification is a kind of ExecutionSpecification representing the execution of a behavior.

Attributes

No additional attributes

Associations

- behavior : Behavior [1] Behavior whose execution is occurring.

Constraints

No additional constraints

Semantics

See “ExecutionSpecification (from BasicInteractions)” on page 464.

Notation

See “ExecutionSpecification (from BasicInteractions)” on page 464.

Rationale

BehaviorExecutionSpecification is introduced to support interactions specifying messages that result from behaviors.

14.3.3 CombinedFragment (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner.

CombinedFragment is a specialization of InteractionFragment.

Attributes

- `interactionOperator : InteractionOperator` Specifies the operation that defines the semantics of this combination of InteractionFragments.

Associations

- `cfragmentGate : Gate[*]` Specifies the gates that form the interface between this CombinedFragment and its surroundings.
- `operand: InteractionOperand[1..*]` The set of operands of the combined fragment.

Constraints

- [1] If the `interactionOperator` is *opt*, *loop*, *break*, or *neg*, there must be exactly one operand.
- [2] The `InteractionConstraint` with `minint` and `maxint` only apply when attached to an `InteractionOperand` where the `interactionOperator` is *loop*.
- [3] If the `interactionOperator` is *break*, the corresponding `InteractionOperand` must cover all Lifelines within the enclosing `InteractionFragment`.
- [4] The interaction operators ‘consider’ and ‘ignore’ can only be used for the `CombineIgnoreFragment` subtype of `CombinedFragment`.
(`(interactionOperator = #consider) or (interactionOperator = #ignore)`) **implies** `oclIsTypeOf(CombineIgnoreFragment)`

Semantics

The semantics of a CombinedFragment is dependent upon the interactionOperator as explained below.

Alternatives

The interactionOperator **alt** designates that the CombinedFragment represents a choice of behavior. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. An implicit true guard is implied if the operand has no guard.

The set of traces that defines a choice is the union of the (guarded) traces of the operands.

An operand guarded by **else** designates a guard that is the negation of the disjunction of all other guards in the enclosing CombinedFragment.

If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing InteractionFragment is executed.

Option

The interactionOperator **opt** designates that the CombinedFragment represents a choice of behavior where either the (sole) operand happens or nothing happens. An option is semantically equivalent to an alternative CombinedFragment where there is one operand with non-empty content and the second operand is empty.

Break

The interactionOperator **break** designates that the CombinedFragment represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing InteractionFragment. A *break* operator with a guard is chosen when the guard is true and the rest of the enclosing Interaction Fragment is ignored. When the guard of the *break* operand is false, the break operand is ignored and the rest of the enclosing InteractionFragment is chosen. The choice between a *break* operand without a guard and the rest of the enclosing InteractionFragment is done non-deterministically.

A CombinedFragment with interactionOperator *break* should cover all Lifelines of the enclosing InteractionFragment.

Parallel

The interactionOperator **par** designates that the CombinedFragment represents a parallel merge between the behaviors of the operands. The EventOccurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

A parallel merge defines a set of traces that describes all the ways that EventOccurrences of the operands may be interleaved without obstructing the order of the EventOccurrences within the operand.

Weak Sequencing

The interactionOperator **seq** designates that the CombinedFragment represents a weak sequencing between the behaviors of the operands.

Weak sequencing is defined by the set of traces with these properties:

1. The ordering of EventOccurrences within each of the operands are maintained in the result.
2. OccurrenceSpecifications on different lifelines from different operands may come in any order.

3. OccurrenceSpecifications on the same lifeline from different operands are ordered such that an EventOccurrence of the first operand comes before that of the second operand.

Thus weak sequencing reduces to a parallel merge when the operands are on disjunct sets of participants. Weak sequencing reduces to strict sequencing when the operands work on only one participant.

Strict Sequencing

The interactionOperator **strict** designates that the CombinedFragment represents a strict sequencing between the behaviors of the operands. The semantics of strict sequencing defines a strict ordering of the operands on the first level within the CombinedFragment with interactionOperator *strict*. Therefore EventOccurrences within contained CombinedFragment will not directly be compared with other EventOccurrences of the enclosing CombinedFragment.

Negative

The interactionOperator **neg** designates that the CombinedFragment represents traces that are defined to be invalid.

The set of traces that defined a CombinedFragment with interactionOperator *negative* is equal to the set of traces given by its (sole) operand, only that this set is a set of invalid rather than valid traces. All InteractionFragments that are different from Negative are considered positive meaning that they describe traces that are valid and should be possible.

Critical Region

The interactionOperator **critical** designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications (on those Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment when determining the set of valid traces. Even though enclosing CombinedFragments may imply that some OccurrenceSpecifications may interleave into the region, such as with *par*-operator, this is prevented by defining a region.

Thus the set of traces of enclosing constructs are restricted by critical regions.

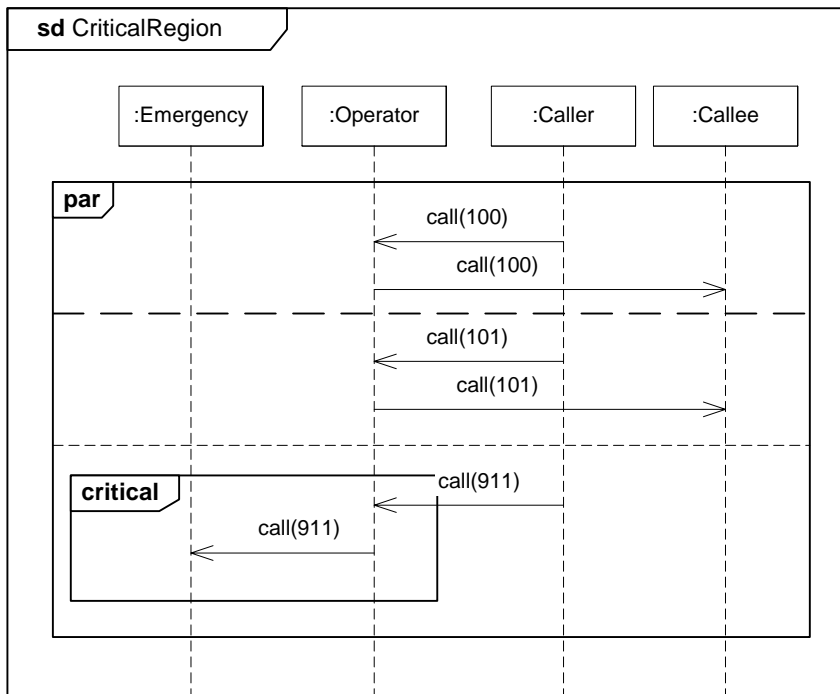


Figure 14.10 - Critical Region

The example, Figure 14.10 shows that the handling of a 911-call must be contiguously handled. The operator must make sure to forward the 911-call before doing anything else. The normal calls, however, can be freely interleaved.

Ignore / Consider

See the semantics of 14.3.4, “ConsiderIgnoreFragment (from Fragments),” on page 458.

Assertion

The interactionOperator **assert** designates that the CombinedFragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace.

Assertions are often combined with Ignore or Consider as shown in Figure 14.24.

Loop

The interactionOperator **loop** designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times.

The Guard may include a lower and an upper number of iterations of the loop as well as a Boolean expression. The semantics is such that a loop will iterate minimum the ‘minint’ number of times (given by the iteration expression in the guard) and at most the ‘maxint’ number of times. After the minimum number of iterations have executed and the boolean expression is false the loop will terminate. The loop construct represents a recursive application of the *seq* operator where the loop operand is sequenced after the result of earlier iterations.

The Semantics of Gates (see also “Gate (from Fragments)” on page 466)

The gates of a CombinedFragment represent the syntactic interface between the CombinedFragment and its surroundings, which means the interface towards other InteractionFragments.

The only purpose of gates is to define the source and the target of Messages.

Notation

The notation for a CombinedFragment in a Sequence Diagram is a solid-outline rectangle. The operator is shown in a pentagon in the upper left corner of the rectangle.

More than one operator may be shown in the pentagon descriptor. This is a shorthand for nesting CombinedFragments. This means that **sd strict** in the pentagon descriptor is the same as two CombinedFragments nested, the outermost with **sd** and the inner with **strict**.

The operands of a CombinedFragment are shown by tiling the graph region of the CombinedFragment using dashed horizontal lines to divide it into regions corresponding to the operands.

Strict

Notationally, this means that the vertical coordinate of the contained fragments is significant throughout the whole scope of the CombinedFragment and not only on one Lifeline. The vertical position of an OccurrenceSpecification is given by the vertical position of the corresponding point. The vertical position of other InteractionFragments is given by the topmost vertical position of its bounding rectangle.

Ignore / Consider

See the notation for “ConsiderIgnoreFragment (from Fragments)” on page 458.

Loop

Textual syntax of the loop operand:

loop[‘(‘ <minint> [‘,’ <maxint>] ‘)’]

<minint> ::= non-negative natural

<maxint> ::= non-negative natural (greater than or equal to <minint> / ‘*’

‘*’ means infinity.

If only <minint> is present, this means that <minint> = <maxint> = <integer>.

If only **loop**, then this means a loop with infinity upper bound and with 0 as lower bound.

Presentation Options for “coregion area”

A notational shorthand for parallel combined fragments are available for the common situation where the order of event occurrences (or other nested fragments) on one Lifeline is insignificant. This means that in a given “coregion” area of a Lifeline all the directly contained fragments are considered separate operands of a parallel combined fragment. See example in Figure 14.12.

Examples

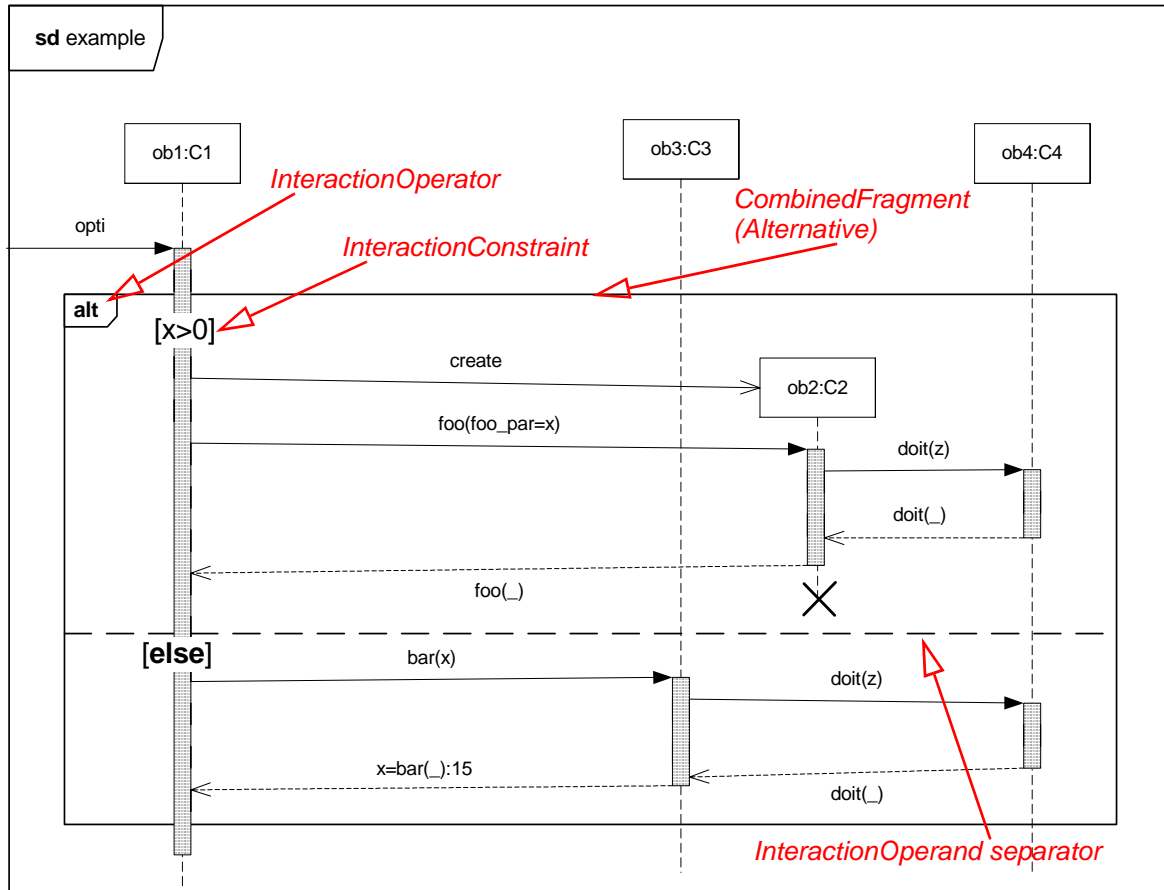


Figure 14.11 - CombinedFragment

Changes from previous UML

This concept was not included in UML 1.x.

14.3.4 ConsiderIgnoreFragment (from Fragments)

Generalizations

- “CombinedFragment (from Fragments)” on page 453

Description

A ConsiderIgnoreFragment is a kind of combined fragment that is used for the consider and ignore cases, which require lists of pertinent messages to be specified.

Attributes

- message : NamedElement [0..*] The set of messages that apply to this fragment.

Constraints

- [1] The interaction operator of a ConsiderIgnoreFragment must be either ‘consider’ or ‘ignore.’
(interactionOperator = #consider) **or** (interactionOperator = #ignore)
- [2] The NamedElements must be of a type of element that identifies a message (e.g., an Operation, Reception, or a Signal).
message->forAll(m | m.ocllsKindOf(Operation) **or** m.ocllsKindOf(Reception) **or** m.ocllsKindOf(Signal))

Semantics

The interactionOperator **ignore** designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are implicitly ignored if they appear in a corresponding execution. Alternatively, one can understand *ignore* to mean that the message types that are ignored can appear anywhere in the traces.

Conversely, the interactionOperator **consider** designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be *ignored*.

Notation

The notation for ConsiderIgnoreFragment is the same as for all CombinedFragments with the keywords **consider** or **ignore** indicating the operator. The list of messages follows the operand enclosed in a pair of braces (curly brackets) according to the following format:

(‘*ignore*’ / ‘*consider*’) {‘<message-name> [‘,’ <message-name>]* ‘’}

Note that ignore and consider can be combined with other types of operations in a single rectangle (as a shorthand for nested rectangles), such as **assert consider** {msgA, msgB}.

Examples

consider {m, s}: showing that only m and s messages are considered significant.

ignore {q,r}: showing that q and r messages are considered insignificant.

Ignore and consider operations are typically combined with other operations such as “**assert consider** {m, s}.”

Figure 14.24 on page 494 shows an example of consider/ignore fragments.

Changes from previous UML

This concept did not exist in UML 1.x.

14.3.5 Continuation (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

A Continuation is a syntactic way to define continuations of different branches of an Alternative CombinedFragment. Continuation is intuitively similar to labels representing intermediate points in a flow of control.

Attributes

- setting : Boolean True when the Continuation is at the end of the enclosing InteractionFragment and False when it is in the beginning.

Constraints

- [1] Continuations with the same name may only cover the same set of Lifelines (within one Classifier).
- [2] Continuations are always global in the enclosing InteractionFragment (e.g., it always covers all Lifelines covered by the enclosing InteractionFragment).
- [3] Continuations always occur as the very first InteractionFragment or the very last InteractionFragment of the enclosing InteractionFragment.

Semantics

Continuations have semantics only in connection with Alternative CombinedFragments and (weak) sequencing.

If an InteractionOperand of an Alternative CombinedFragment ends in a Continuation with name (say) X, only InteractionFragments starting with the Continuation X (or no continuation at all) can be appended.

Notation

Continuations are shown with the same symbol as States, but they may cover more than one Lifeline.

Continuations may also appear on flowlines of Interaction Overview Diagrams.

Continuations that are alone in an InteractionFragment are considered to be at the end of the enclosing InteractionFragment.

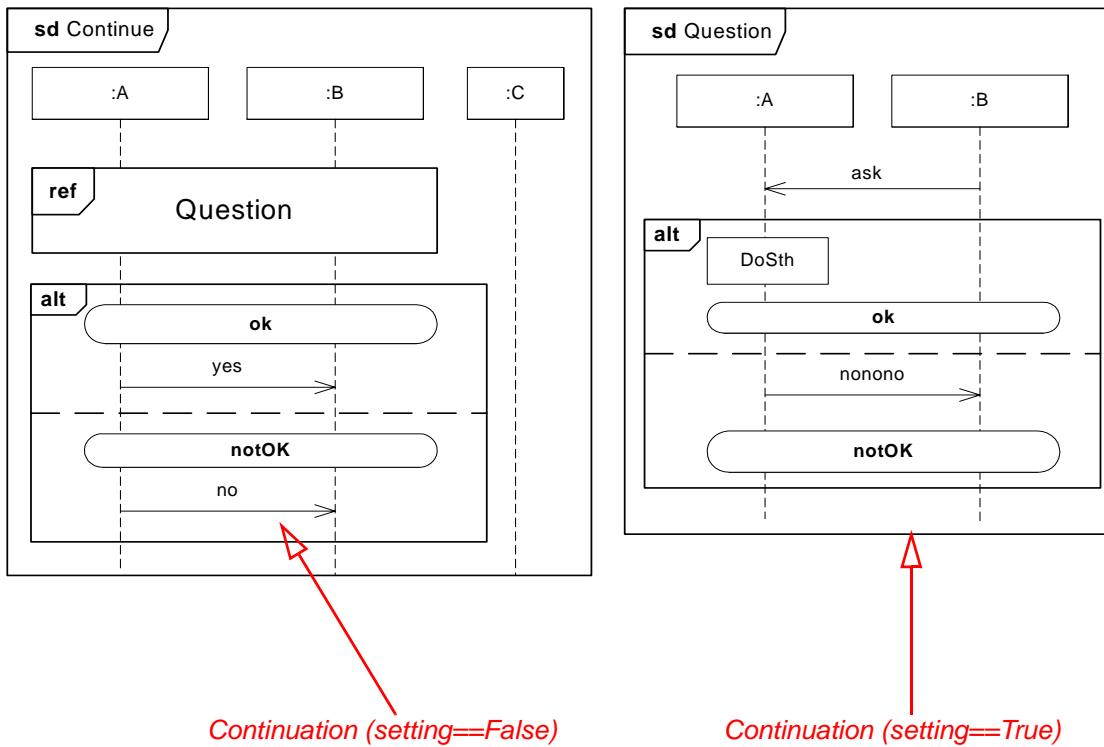


Figure 14.12 - Continuation

The two diagrams in Figure 14.12 are together equivalent to the diagram in Figure 14.13.

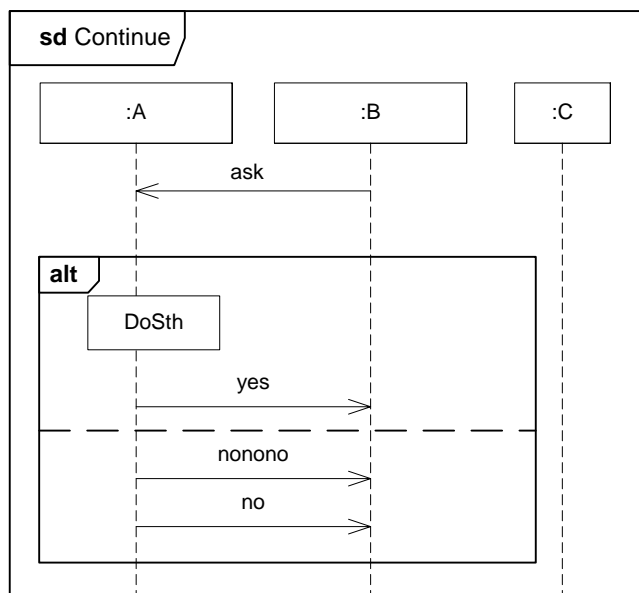


Figure 14.13 - Continuation interpretation

14.3.6 CreationEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 428

Description

A CreationEvent models the creation of an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A creation event represents the creation of an instance. It may result in the subsequent execution of initializations as well as the invocation of the specified classifier behavior (see “Common Behaviors” on page 407).

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.7 DestructionEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 428

Description

A DestructionEvent models the destruction of an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

- [1] No other OccurrenceSpecifications may appear below an OccurrenceSpecification that references a DestructionEvent on a given Lifeline in an InteractionOperand.

Semantics

A destruction event represents the destruction of the instance described by the lifeline containing the OccurrenceSpecification that references the destruction event. It may result in the subsequent destruction of other instances that this instance owns by composition (see “Common Behaviors” on page 407).

Notation

The DestructionEvent is depicted by a cross in the form of an X at the bottom of a Lifeline.



Figure 14.14 - DestructionEvent symbol

See example in Figure 14.11.

Changes from previous UML

DestructionEvent is new in UML 2.0.

14.3.8 ExecutionEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 428

Description

An ExecutionEvent models the start or finish of an execution occurrence.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An execution event represents the start or finish of the execution of an action or a behavior.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.9 ExecutionOccurrenceSpecification (from BasicInteractions)

Generalizations

- “OccurrenceSpecification (from BasicInteractions)” on page 481

Description

An ExecutionOccurrenceSpecification represents moments in time at which actions or behaviors start or finish.

Attributes

No additional attributes

Associations

- event : ExecutionEvent [1] Redefines the event referenced to be restricted to an execution event.
- execution : ExecutionSpecification [1] References the execution specification describing the execution that is started or finished at this execution event.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.10 ExecutionSpecification (from BasicInteractions)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

An ExecutionSpecification is a specification of the execution of a unit of behavior or action within the Lifeline. The duration of an ExecutionSpecification is represented by two ExecutionOccurrenceSpecifications, the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification.

Associations

- **start** : OccurrenceSpecification[1] References the OccurrenceSpecification that designates the start of the Action or Behavior.
- **finish**: OccurrenceSpecification[1] References the OccurrenceSpecification that designates the finish of the Action or Behavior.

Constraints

- [1] The startEvent and the finishEvent must be on the same Lifeline.
start.lifeline = finish.lifeline

Semantics

The trace semantics of Interactions merely see an Execution as the trace <start, finish>. There may be occurrences between these. Typically the start occurrence and the finish occurrence will represent OccurrenceSpecifications such as a receive OccurrenceSpecification (of a Message) and the send OccurrenceSpecification (of a reply Message).

Notation

ExecutionOccurrences are represented as thin rectangles (grey or white) on the lifeline (see “Lifeline (from BasicInteractions, Fragments)” on page 475).

We may also represent an ExecutionSpecification by a wider labeled rectangle, where the label usually identifies the action that was executed. An example of this can be seen in Figure 14.13 on page 461.

ExecutionSpecifications that refer to atomic actions such as reading attributes of a Signal (conveyed by the Message), the Action symbol may be associated with the reception OccurrenceSpecification with a line in order to emphasize that the whole Action is associated with only one OccurrenceSpecification (and start and finish associations refer the very same OccurrenceSpecification).

Overlapping execution occurrences on the same lifeline are represented by overlapping rectangles as shown in Figure 14.15.

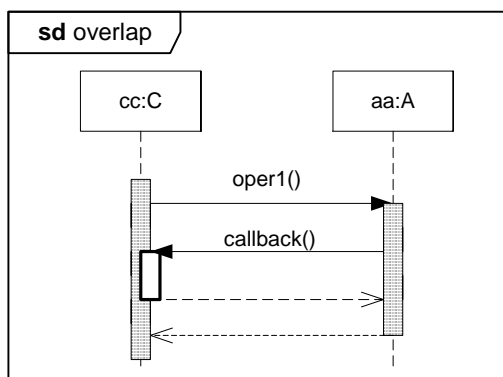


Figure 14.15 - Overlapping execution occurrences

14.3.11 Gate (from Fragments)

Generalizations

- “MessageEnd (from BasicInteractions)” on page 479

Description

A Gate is a connection point for relating a Message outside an InteractionFragment with a Message inside the InteractionFragment.

Gate is a specialization of MessageEnd.

Gates are connected through Messages. A Gate is actually a representative of an OccurrenceSpecification that is not in the same scope as the Gate.

Gates play different roles: we have formal gates on Interactions, actual gates on InteractionUses, expression gates on CombinedFragments.

Constraints

- [1] The message leading to/from an actualGate of an InteractionUse must correspond to the message leading from/to the formalGate with the same name of the Interaction referenced by the InteractionUse.
- [2] The message leading to/from an (expression) Gate within a CombinedFragment must correspond to the message leading from/to the CombinedFragment on its outside.

Semantics

The gates are named either explicitly or implicitly. Gates may be identified either by name (if specified), or by a constructed identifier formed by concatenating the direction of the message and the message name (e.g., *out_CardOut*). The gates and the messages between gates have one purpose, namely to establish the concrete sender and receiver for every message.

Notation

Gates are just points on the frame, the ends of the messages. They may have an explicit name (see Figure 14.19).

The same gate may appear several times in the same or different diagrams.

14.3.12 GeneralOrdering (from BasicInteractions)

Generalizations

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

Description

A GeneralOrdering represents a binary relation between two OccurrenceSpecifications, to describe that one OccurrenceSpecification must occur before the other in a valid trace. This mechanism provides the ability to define partial orders of OccurrenceSpecifications that may otherwise not have a specified order.

A GeneralOrdering is a specialization of NamedElement.

A GeneralOrdering may appear anywhere in an Interaction, but only between OccurrenceSpecifications.

Associations

- before: OccurrenceSpecification[1] The OccurrenceSpecification referred comes before the OccurrenceSpecification referred by *after*.
- after: OccurrenceSpecification[1] The OccurrenceSpecification referred comes after the OccurrenceSpecification referred by *before*.

Semantics

A GeneralOrdering is introduced to restrict the set of possible sequences. A partial order of OccurrenceSpecifications is defined by a set of GeneralOrdering.

Notation

A GeneralOrdering is shown by a dotted line connecting the two OccurrenceSpecifications. The direction of the relation from the *before* to the *after* is given by an arrowhead placed somewhere in the middle of the dotted line (i.e., not at the endpoint).

14.3.13 Interaction (from BasicInteraction, Fragments)

Generalizations

- “Behavior (from BasicBehaviors)” on page 416
- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

An interaction is a unit of behavior that focuses on the observable exchange of information between ConnectableElements.

An Interaction is a specialization of InteractionFragment and of Behavior.

Associations

- formalGate: Gate[*] Specifies the gates that form the message interface between this Interaction and any InteractionUses that reference it.
- lifeline: LifeLine[0..*] Specifies the participants in this Interaction.
- event: MessageEnd[*] MessageEnds (e.g., OccurrenceSpecifications or Gates) owned by this Interaction.
- message: Message[*] The Messages contained in this Interaction.
- fragment: InteractionFragment[*] The ordered set of fragments in the Interaction.
- action: Action[*] Actions owned by the Interaction. See “ActionExecutionSpecification (from BasicInteractions)” on page 452.

Semantics

Interactions are units of behavior of an enclosing Classifier. Interactions focus on the passing of information with Messages between the ConnectableElements of the Classifier.

The semantics of an Interaction is given as a pair of sets of traces. The two trace sets represent valid traces and invalid traces. The union of these two sets need not necessarily cover the whole universe of traces. The traces that are not included are not described by this Interaction at all, and we cannot know whether they are valid or invalid.

A trace is a sequence of event occurrences, each of which is described by an OccurrenceSpecification in a model. The semantics of Interactions are compositional in the sense that the semantics of an Interaction is mechanically built from the semantics of its constituent InteractionFragments. The constituent InteractionFragments are ordered and combined by the seq operation (weak sequencing) as explained in “Weak Sequencing” on page 454.

The invalid set of traces are associated only with the use of a Negative CombinedInteraction. For simplicity we describe only valid traces for all other constructs.

As Behavior an Interaction is generalizable and redefineable. Specializing an Interaction is simply to add more traces to those of the original. The traces defined by the specialization is combined with those of the inherited Interaction with a union.

The classifier owning an Interaction may be specialized, and in the specialization the Interaction may be redefined. Redefining an Interaction simply means to exchange the redefining Interaction for the redefined one, and this exchange takes effect also for InteractionUses within the supertype of the owner. This is similar to redefinition of other kinds of Behavior.

Basic trace model: The semantics of an Interaction is given by a pair $[P, I]$ where P is the set of valid traces and I is the set of invalid traces. $P \cup I$ need not be the whole universe of traces.

A trace is a sequence of event occurrences denoted $\langle e_1, e_2, \dots, e_n \rangle$.

An event occurrence will also include information about the values of all relevant objects at this point in time.

Each construct of Interactions (such as CombinedFragments of different kinds) are expressed in terms of how it relates to a pair of sets of traces. For simplicity we normally refer only to the set of valid traces as these traces are those mostly modeled.

Two Interactions are equivalent if their pair of trace-sets are equal.

Relation of trace model to execution model: In Chapter 13, “Common Behaviors” we find an Execution model, and this is how the Interactions Trace Model relates to the Execution model.

An Interaction is an Emergent Behavior.

An InvocationOccurrence in the Execution model corresponds with an (event) Occurrence in a trace. Occurrences are modeled in an Interaction by OccurrenceSpecifications. Normally in Interaction the action leading to the invocation as such is not described (such as the sending action). However, if it is desirable to go into details, a Behavior (such as an Activity) may be associated with an OccurrenceSpecification. An occurrence in Interactions is normally interpreted to take zero time. Duration is always between occurrences.

Likewise a ReceiveOccurrence in the Execution model is modeled by an OccurrenceSpecification. Similarly the detailed actions following immediately from this reception is often omitted in Interactions, but may also be described explicitly with a Behavior associated with that OccurrenceSpecification.

A Request in the Execution model is modeled by the Message in Interactions.

An Execution in the Execution model is modeled by an ExecutionSpecification in Interactions. An Execution is defined in the trace by two Occurrences, one at the start and one at the end. This corresponds to the StartOccurrence and the CompletionOccurrence of the Execution model.

Notation

The notation for an Interaction in a Sequence Diagram is a solid-outline rectangle. The keyword **sd** followed by the Interaction name and parameters is in a pentagon in the upper left corner of the rectangle. The notation within this rectangular frame comes in several forms: Sequence Diagrams, Communication Diagrams, Interaction Overview Diagrams, and Timing Diagrams.

The notation within the pentagon descriptor follows the general notation for the name of Behaviors. In addition the Interaction Overview Diagrams may include a list of Lifelines through a lifeline-clause as shown in Figure 14.28. The list of lifelines is simply a listing of the Lifelines involved in the Interaction. An Interaction Overview Diagram does not in itself show the involved lifelines even though the lifelines may occur explicitly within inline Interactions in the graph nodes.

An Interaction diagram may also include definitions of local attributes with the same syntax as attributes in general are shown within class symbol compartments. These attribute definitions may appear near the top of the diagram frame or within note symbols at other places in the diagram.

Please refer to Section 14.4 to see examples of notation for Interactions.

Examples

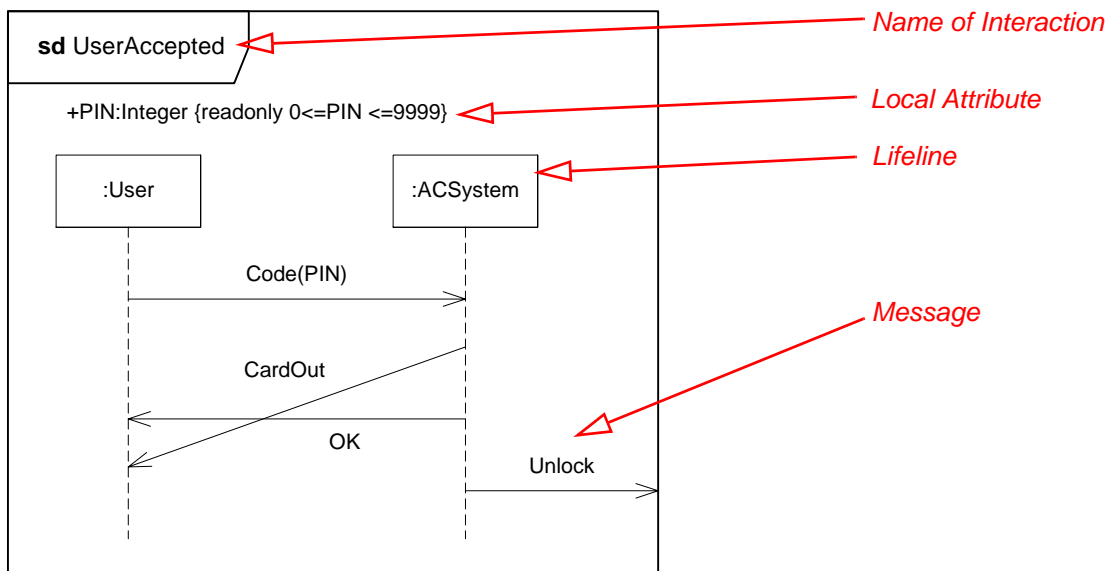


Figure 14.16 - An example of an Interaction in the form of a Sequence Diagram

The example in Figure 14.16 shows three messages communicated between two (anonymous) lifelines of types *User* and *ACSystem*. The message *CardOut* overtakes the message *OK* in the way that the receiving event occurrences are in the opposite order of the sending EventOccurrences. Such communication may occur when the messages are asynchronous. Finally a fourth message is sent from the *ACSystem* to the environment through a gate with implicit name *out_Unlock*. The local attribute *PIN* of *UserAccepted* is declared near the diagram top. It could have been declared in a Note somewhere else in the diagram.

Changes from previous UML

Interactions are now contained within Classifiers and not only within Collaborations. Their participants are modeled by Lifelines instead of ClassifierRoles.

14.3.14 InteractionConstraint (from Fragments)

Generalizations

- “Constraint (from Kernel)” on page 54

Description

An InteractionConstraint is a boolean expression that guards an operand in a CombinedFragment.

InteractionConstraint is a specialization of Constraint.

Furthermore the InteractionConstraint contains two expressions designating the minimum and maximum number of times a loop CombinedFragment should execute.

Associations

- minint: ValueSpecification[0..1] The minimum number of iterations of a loop.
- maxint: ValueSpecification[0..1] The maximum number of iterations of a loop.

Constraints

- [1] The dynamic variables that take part in the constraint must be owned by the ConnectableElement corresponding to the covered Lifeline.
- [2] The constraint may contain references to global data or write-once data.
- [3] Minint/maxint can only be present if the InteractionConstraint is associated with the operand of a loop CombinedFragment.
- [4] If minint is specified, then the expression must evaluate to a non-negative integer.
- [5] If maxint is specified, then the expression must evaluate to a positive integer.
- [6] If maxint is specified, then minint must be specified and the evaluation of maxint must be \geq the evaluation of minint.

Semantics

InteractionConstraints are always used in connection with CombinedFragments, see “CombinedFragment (from Fragments)” on page 453.

Notation

An InteractionConstraint is shown in square brackets covering the lifeline where the first event occurrence will occur, positioned above that event, in the containing Interaction or InteractionOperand.

<interactionconstraint> ::= [[' (<Boolean-expression' / 'else') ']]

When the InteractionConstraint is omitted, true is assumed.

Please refer to an example of InteractionConstraints in Figure 14.11 on page 458.

14.3.15 InteractionFragment (from BasicInteractions, Fragments)

Generalizations

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

Description

InteractionFragment is an abstract notion of the most general interaction unit. An interaction fragment is a piece of an interaction. Each interaction fragment is conceptually like an interaction by itself.

InteractionFragment is an abstract class and a specialization of NamedElement.

Associations

- enclosingOperand: InteractionOperand[0..1] The operand enclosing this InteractionFragment (they may nest recursively).
- covered : Lifeline[*] References the Lifelines that the InteractionFragment involves.
- generalOrdering:GeneralOrdering[*] The general ordering relationships contained in this fragment.
- enclosingInteraction: Interaction[0..1] The Interaction enclosing this InteractionFragment.

Semantics

The semantics of an InteractionFragment is a pair of set of traces. See “Interaction (from BasicInteraction, Fragments)” for explanation of how to calculate the traces.

Notation

There is no general notation for an InteractionFragment. The specific subclasses of InteractionFragment will define their own notation.

Changes from previous UML

This concept did not appear in UML 1.x.

14.3.16 InteractionOperand (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471
- “Namespace (from Kernel)” on page 95

Description

An InteractionOperand is contained in a CombinedFragment. An InteractionOperand represents one operand of the expression given by the enclosing CombinedFragment.

An InteractionOperand is an InteractionFragment with an optional guard expression. An InteractionOperand may be guarded by an InteractionConstraint. Only InteractionOperands with a guard that evaluates to true at this point in the interaction will be considered for the production of the traces for the enclosing CombinedFragment.

InteractionOperand contains an ordered set of InteractionFragments.

In Sequence Diagrams these InteractionFragments are ordered according to their geometrical position vertically. The geometrical position of the InteractionFragment is given by the topmost vertical coordinate of its contained EventOccurrences or symbols.

Associations

- fragment: InteractionFragment[*] The fragments of the operand.
- guard: InteractionConstraint[0..1] Constraint of the operand.

Constraints

- [1] The guard must be placed directly prior to (above) the EventOccurrence that will become the first EventOccurrence within this InteractionOperand.
- [2] The guard must contain only references to values local to the Lifeline on which it resides, or values global to the whole Interaction (See “InteractionConstraint (from Fragments)” on page 470).

Semantics

Only InteractionOperands with true guards are included in the calculation of the semantics. If no guard is present, this is taken to mean a true guard.

The semantics of an InteractionOperand is given by its constituent InteractionFragments combined by the implicit *seq* operation. The seq operator is described in “CombinedFragment (from Fragments)” on page 453.

Notation

InteractionOperands are separated by a dashed horizontal line. The InteractionOperands together make up the framed CombinedFragment.

Within an InteractionOperand of a Sequence Diagram the order of the InteractionFragments are given simply by the topmost vertical position.

See Figure 14.11 for examples of InteractionOperand.

14.3.17 InteractionOperator (from Fragments)

Generalizations

None

Description

Interaction Operator is an enumeration designating the different kinds of operators of CombinedFragments. The InteractionOperand defines the type of operator of a CombinedFragment. The literal values of this enumeration are:

- alt
- opt
- par
- loop
- critical
- neg

- assert
- strict
- seq
- ignore
- consider

Semantics

The value of the interactionOperator is significant for the semantics of “CombinedFragment (from Fragments)” on page 453.

Notation

The value of the InteractionOperand is given as text in a small compartment in the upper left corner of the CombinedFragment frame. See Figure 14.11 on page 458 for examples of InteractionOperator.

14.3.18 InteractionUse (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

An InteractionUse refers to an Interaction. The InteractionUse is a shorthand for copying the contents of the referred Interaction where the InteractionUse is. To be accurate the copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones.

It is common to want to share portions of an interaction between several other interactions. An InteractionUse allows multiple interactions to reference an interaction that represents a common portion of their specification.

Description

InteractionUse is a specialization of InteractionFragment.

An InteractionUse has a set of actual gates that must match the formal gates of the referenced Interaction.

Associations

- refersTo: Interaction[1] Refers to the Interaction that defines its meaning.
- argument:InputPin[*] The actual arguments of the Interaction.
- actualGate:Gate[*] The actual gates of the InteractionUse.

Constraints

- [1] Actual Gates of the InteractionUse must match Formal Gates of the referred Interaction. Gates match when their names are equal.
- [2] The InteractionUse must cover all Lifelines of the enclosing Interaction that appear within the referred Interaction.
- [3] The arguments of the InteractionUse must correspond to parameters of the referred Interaction.

- [4] The arguments must only be constants, parameters of the enclosing Interaction or attributes of the classifier owning the enclosing Interaction.

Semantics

The semantics of the InteractionUse is the set of traces of the semantics of the referred Interaction where the gates have been resolved as well as all generic parts having been bound such as the arguments substituting the parameters.

Notation

The InteractionUse is shown as a CombinedFragment symbol where the operator is called *ref*. The complete syntax of the name (situated in the InteractionUse area) is:

```
<name> ::= [<attribute-name> '=' ] [ <collaboration-use> '.' ] <interaction-name>
          [ '(' <io-argument> [ ',' <io-oargument> ]* ')' ] [ ':' <return-value> ]
<io-argument> ::= <in-argument> / 'out' <out-argument> ]
```

The *<attribute-name>* refers to an attribute of one of the lifelines in the Interaction.

<collaboration-use> is an identification of a collaboration use that binds lifelines of a collaboration. The interaction name is in that case within that collaboration. See example of collaboration uses in Figure 14.25.

The *io-arguments* are most often arguments of IN-parameters. If there are OUT- or INOUT-parameters and the output value is to be described, this can be done following an **out** keyword.

The syntax of *argument* is explained in the notation section of Messages (“Message (from BasicInteractions)” on page 477).

If the InteractionUse returns a value, this may be described following a colon at the end of the clause.

Examples

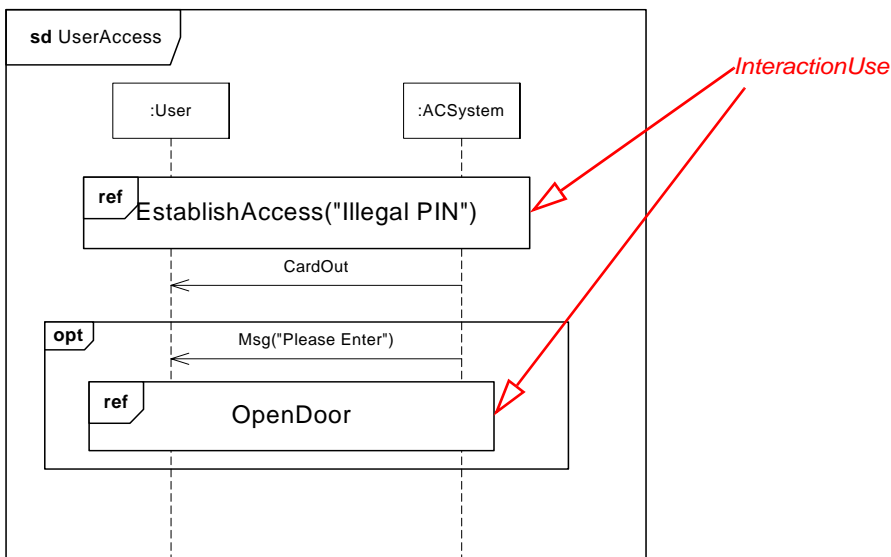


Figure 14.17 - InteractionUse

In Figure 14.17 we show an *InteractionUse* referring the Interaction *EstablishAccess* with (input) argument “Illegal PIN.” Within the optional *CombinedFragment* there is another *InteractionUse* without arguments referring *OpenDoor*.

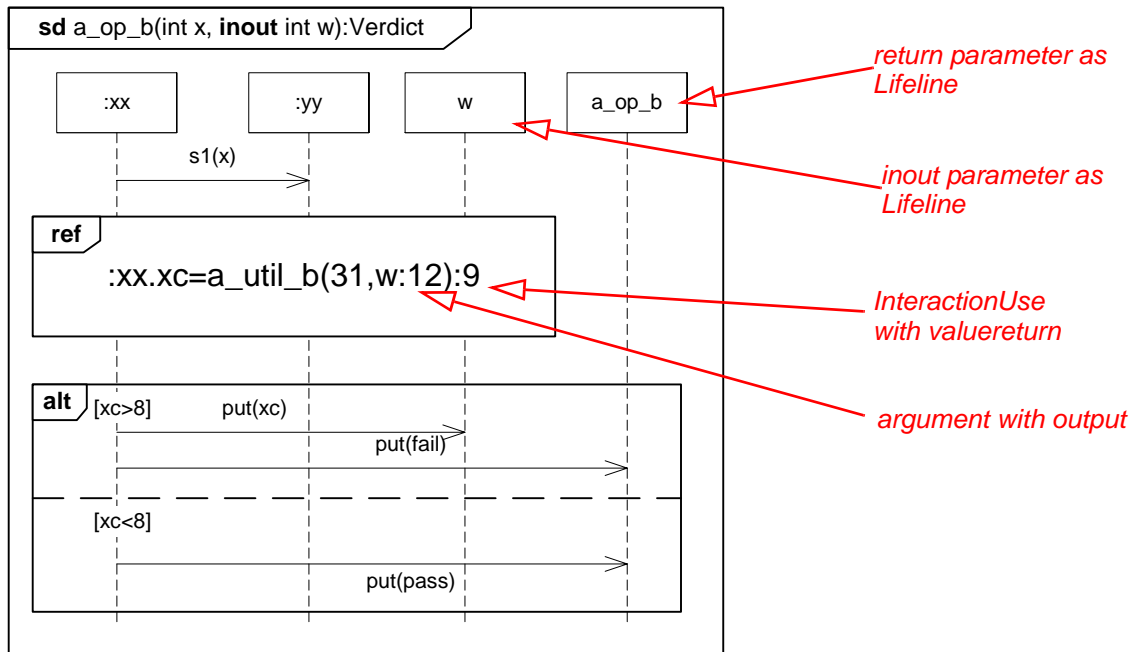


Figure 14.18 - InteractionUse with value return

In Figure 14.18 we have a more advanced Interaction that models a behavior returning a *Verdict* value. The return value from the Interaction is shown as a separate Lifeline *a_op_b*. Inside the Interaction there is an *InteractionUse* referring *a_util_b* with value return to the attribute *xc* of *:xx* with the value 9, and with inout parameter where the argument is *w* with returning out-value 12.

Changes from previous UML

InteractionUse was not a concept in UML 1.x.

14.3.19 Lifeline (from BasicInteractions, Fragments)

Generalizations

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

Description

A lifeline represents an individual participant in the Interaction. While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity.

Lifeline is a specialization of *NamedElement*.

If the referenced *ConnectableElement* is multivalued (i.e. has a multiplicity > 1), then the Lifeline may have an expression (the ‘selector’) that specifies which particular part is represented by this Lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued *ConnectableElement* is chosen.

Associations

- selector : Expression[0..1] If the referenced ConnectableElement is multivalued, then this specifies the specific individual part within that set.
- interaction: Interaction[1] References the Interaction enclosing this Lifeline.
- represents: ConnectableElement[0..1] References the ConnectableElement within the classifier that contains the enclosing interaction.
- decomposedAs : PartDecomposition[0..1] References the Interaction that represents the decomposition.

Constraints

- [1] If two (or more) InteractionUses within one Interaction, refer to Interactions with common Lifelines, those Lifelines must also appear in the Interaction with the InteractionUses. By ‘common Lifelines’ we mean Lifelines with the same selector and represents associations.
- [2] The selector for a Lifeline must only be specified if the referenced Part is multivalued.
(self.selector->isEmpty() **implies not** self.represents.isMultivalued()) **or**
(**not** self.selector->isEmpty() **implies** self.represents.isMultivalued())
- [3] The classifier containing the referenced ConnectableElement must be the same classifier, or an ancestor, of the classifier that contains the interaction enclosing this lifeline.
if (represents->notEmpty()) **then**
 (**if** selector->notEmpty() **then** represents.isMultivalued() **else not** represents.isMultivalued())

Semantics

The order of OccurrenceSpecifications along a Lifeline is significant denoting the order in which these OccurrenceSpecifications will occur. The absolute distances between the OccurrenceSpecifications on the Lifeline are, however, irrelevant for the semantics.

The semantics of the Lifeline (within an Interaction) is the semantics of the Interaction selecting only OccurrenceSpecifications of this Lifeline.

Notation

A Lifeline is shown using a symbol that consists of a rectangle forming its “head” followed by a vertical line (which may be dashed) that represents the lifetime of the participant. Information identifying the lifeline is displayed inside the rectangle in the following format:

```
<lifelineident> ::= ([<connectable-element-name>[['<selector> ']] [: <class_name>] [decomposition]) | 'self'  
<selector> ::= <expression>  
<decomposition> ::= 'ref' <interactionident> ['strict']
```

where <class-name> is the type referenced by the represented ConnectableElement. Note that, although the syntax allows it, <lifelineident> cannot be empty.

The Lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Often the head is a white rectangle containing the name.

If the name is the keyword **self**, then the lifeline represents the object of the classifier that encloses the Interaction that owns the Lifeline. Ports of the enclosure may be shown separately even when *self* is included.

To depict method activations we apply a thin grey or white rectangle that covers the Lifeline line.

Examples

See Figure 14.16 where the Lifelines are pointed to.

See Figure 14.11 to see method activations.

Changes from previous UML

Lifelines are basically the same concept as before in UML 1.x.

14.3.20 Message (from BasicInteractions)

Generalizations

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

Description

A Message defines a particular communication between Lifelines of an Interaction.

A Message is a NamedElement that defines one specific kind of communication in an Interaction. A communication can be, for example, raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication given by the dispatching ExecutionSpecification, but also the sender and the receiver.

A Message associates normally two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification.

Attributes

- messageKind:MessageKind The derived kind of the Message (*complete*, *lost*, *found*, or *unknown*).
- messageSort:MessageSort The sort of communication reflected by the Message.

Associations

- interaction:Interaction[1] The enclosing Interaction owning the Message.
- sendEvent : MessageEnd[0..1] References the Sending of the Message.
- receiveEvent: MessageEnd[0..1] References the Receiving of the Message.
- connector: Connector[0..1] The Connector on which this Message is sent.
- argument:ValueSpecification[*] The arguments of the Message.
- /signature:NamedElement[0..1] The definition of the type or signature of the Message (depending on its kind). The associated named element is derived from the message end that constitutes the sending or receiving message event. If both a sending event and a receiving message event are present, the signature is obtained from the sending event.

Constraints

- [1] If the sending MessageEvent and the receiving MessageEvent of the same Message are on the same Lifeline, the sending MessageEvent must be ordered before the receiving MessageEvent.
- [2] The signature must either refer to an Operation (in which case messageSort is either synchCall or asynchCall) or a Signal (in which case messageSort is asynchSignal). The name of the NamedElement referenced by signature must be the same as that of the Message.

- [3] In the case when the Message signature is an Operation, the arguments of the Message must correspond to the parameters of the Operation. A Parameter corresponds to an Argument if the Argument is of the same Class or a specialization of that of the Parameter.
- [4] In the case when the Message signature is a Signal, the arguments of the Message must correspond to the attributes of the Signal. A Message Argument corresponds to a Signal Attribute if the Argument is of the same Class or a specialization of that of the Attribute.
- [5] Arguments of a Message must only be:
- i) attributes of the sending lifeline.
 - ii) constants.
 - iii) symbolic values (which are wildcard values representing any legal value).
 - iv) explicit parameters of the enclosing Interaction.
 - v) attributes of the class owning the Interaction.
- [6] Messages cannot cross boundaries of CombinedFragments or their operands.
- [7] If the MessageEnds are both OccurrenceSpecifications, then the connector must go between the Parts represented by the Lifelines of the two MessageEnds.

Semantics

The semantics of a *complete* Message is simply the trace <sendEvent, receiveEvent>.

A *lost* message is a message where the sending event occurrence is known, but there is no receiving event occurrence. We interpret this to be because the message never reached its destination. The semantics is simply the trace <sendEvent>.

A *found* message is a message where the receiving event occurrence is known, but there is no (known) sending event occurrence. We interpret this to be because the origin of the message is outside the scope of the description. This may for example be noise or other activity that we do not want to describe in detail. The semantics is simply the trace <receiveEvent>.

A Message reflects either an Operation call and start of execution - or a sending and reception of a Signal.

When a Message represents an Operation invocation, the arguments of the Message are the arguments of the CallOperationAction on the sending Lifeline as well as the arguments of the CallEvent occurrence on the receiving Lifeline.

When a Message represents a Signal, the arguments of the Message are the arguments of the SendAction on the sending Lifeline and on the receiving Lifeline the arguments are available in the SignalEvent.

If the Message represents a CallAction, there will normally be a reply message from the called Lifeline back to the calling lifeline before the calling Lifeline will proceed.

Notation

A message is shown as a line from the sender message end to the receiver message end. The form of the line or arrowhead reflect properties of the message:

- Asynchronous Messages have an open arrow head.
- Synchronous Messages typically represent method calls and are shown with a filled arrow head. The reply message from a method has a dashed line.
- Object creation Message has a dashed line with an open arrow.
- Lost Messages are described as a small black circle at the arrow end of the Message.

- Found Messages are described as a small black circle at the starting end of the Message.
- On Communication Diagrams, the Messages are decorated by a small arrow along the connector close to the Message name and sequence number in the direction of the Message.

Syntax for the Message name is the following:

```
<messageident> ::= ([<attribute> '=' ] <signal-or-operation-name> ['(' [<argument> [','<argument>]* ')']
                    [':' <return-value>] ] | '*'
<argument> ::= (<[parameter-name> '=' ] <argument-value>) | (<attribute> '=' <out-parameter-name>
                    [':' <argument-value>] | '-'
```

Messageident equaling '*' is a shorthand for more complex alternative CombinedFragment to represent a message of any type. This is to match asterisk triggers in State Machines.

Return-value and attribute assignment are used only for reply messages. Attribute assignment is a shorthand for including the Action that assigns the return-value to that attribute. This holds both for the possible return value of the message (the return value of the associated operation), and the out values of (in)out parameters.

When the argument list contains only argument-values, all the parameters must be matched either by a value or by a dash (-). If parameter-names are used to identify the argument-value, then arguments may freely be omitted. Omitted parameters get an unknown argument-value.

Examples

In Figure 14.16 we see only asynchronous Messages. Such Messages may overtake each other.

In Figure 14.11 we see method calls that are synchronous accompanied by replies. We also see a Message that represents the creation of an object.

In Figure 14.27 we see how Messages are denoted in Communication Diagrams.

Examples of syntax:

```
mymessage(14, -, 3.14, "hello")    // second argument is undefined
v=mymsg(16, variab):96             // this is a reply message carrying the return value 96 assigning it to v
mymsg(myint=16)                   // the input parameter 'myint' is given the argument value 16
```

See Figure 14.11 for a number of different applications of the textual syntax of message identification.

Changes from previous UML

Messages may have Gates on either end.

14.3.21 MessageEnd (from BasicInteractions)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 93.

Description

A MessageEnd is an abstract NamedElement that represents what can occur at the end of a Message.

Associations

- message : Message [1] References a Message.

Semantics

Subclasses of MessageEnd define the specific semantics appropriate to the concept they represent.

14.3.22 MessageKind (from BasicInteractions)

This is an enumerated type that identifies the type of message.

Generalizations

None

Description

MessageSort is an enumeration of the following values:

- complete = sendEvent and receiveEvent are present.
- lost = sendEvent present and receiveEvent absent.
- found = sendEvent absent and receiveEvent present.
- unknown = sendEvent and receiveEvent absent (should not appear).

14.3.23 MessageOccurrenceSpecification (from BasicInteractions)

Generalizations

- “MessageEnd (from BasicInteractions)” on page 479
- “OccurrenceSpecification (from BasicInteractions)” on page 481

Description

Specifies the occurrence of message events, such as sending and receiving of signals or invoking or receiving of operation calls. A message occurrence specification is a kind of message end. Messages are generated either by synchronous operation calls or asynchronous signal sends. They are received by the execution of corresponding accept event actions.

Attributes

No additional attributes

Associations

- event : MessageEvent [1] Redefines the event referenced to be restricted to a message event.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.24 MessageSort (from BasicInteractions)

This is an enumerated type that identifies the type of communication action that was used to generate the message.

Generalizations

None

Description

MessageSort is an enumeration of the following values:

- synchCall - The message was generated by a synchronous call to an operation.
- asynchCall - The message was generated by an asynchronous call to an operation (i.e., a CallAction with “isSynchronous = false”).
- asynchSignal - The message was generated by an asynchronous send action.

14.3.25 OccurrenceSpecification (from BasicInteractions)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

An OccurrenceSpecification is the basic semantic unit of Interactions. The sequences of occurrences specified by them are the meanings of Interactions.

OccurrenceSpecifications are ordered along a Lifeline.

The namespace of an OccurrenceSpecification is the Interaction in which it is contained.

Associations

- event : Event [1] References a specification of the occurring event.
- covered: Lifeline[1] References the Lifeline on which the OccurrenceSpecification appears.
 Redefines InteractionFragment.covered.
- toBefore:GeneralOrdering[*] References the GeneralOrderings that specify EventOccurrences that must occur before this OccurrenceSpecification.
- toAfter: GeneralOrdering[*] References the GeneralOrderings that specify EventOccurrences that must occur after this OccurrenceSpecification.

Semantics

The semantics of an OccurrenceSpecification is just the trace of that single OccurrenceSpecification.

The understanding and deeper meaning of the OccurrenceSpecification is dependent upon the associated Message and the information that it conveys.

Notation

OccurrenceSpecifications are merely syntactic points at the ends of Messages or at the beginning/end of an ExecutionSpecification.

Examples

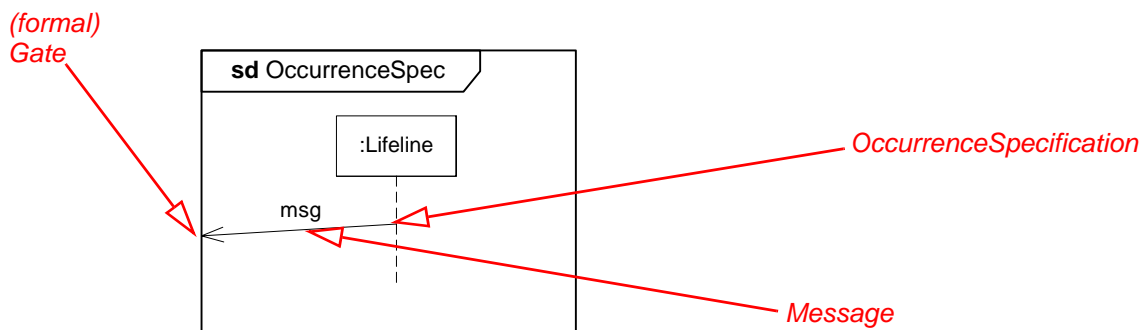


Figure 14.19 - OccurrenceSpecification

14.3.26 PartDecomposition (from Fragments)

Generalizations

- “InteractionUse (from Fragments)” on page 473

Description

PartDecomposition is a description of the internal interactions of one Lifeline relative to an Interaction.

A Lifeline has a class associated as the type of the ConnectableElement that the Lifeline represents. That class may have an internal structure and the PartDecomposition is an Interaction that describes the behavior of that internal structure relative to the Interaction where the decomposition is referenced.

A PartDecomposition is a specialization of InteractionUse. It associates with the ConnectableElement that it decomposes.

Constraints

- [1] PartDecompositions apply only to Parts that are Parts of Internal Structures not to Parts of Collaborations.
- [2] Assume that within Interaction X, Lifeline L is of class C and decomposed to D. Within X there is a sequence of constructs along L (such constructs are CombinedFragments, InteractionUse and (plain) OccurrenceSpecifications). Then a corresponding sequence of constructs must appear within D, matched one-to-one in the same order.
 - i) CombinedFragment covering L are matched with an extra-global CombinedFragment in D.
 - ii) An InteractionUse covering L are matched with a global (i.e., covering all Lifelines) InteractionUse in D.
 - iii) A plain OccurrenceSpecification on L is considered an actualGate that must be matched by a formalGate of D.

- [3] Assume that within Interaction X, Lifeline L is of class C and decomposed to D. Assume also that there is within X an InteractionUse (say) U that covers L. According to the constraint above U will have a counterpart CU within D. Within the Interaction referenced by U, L should also be decomposed, and the decomposition should reference CU. (This rule is called commutativity of decomposition.)

Semantics

Decomposition of a lifeline within one Interaction by an Interaction (owned by the type of the Lifeline's associated ConnectableElement), is interpreted exactly as an InteractionUse. The messages that go into (or go out from) the decomposed lifeline are interpreted as actual gates that are matched by corresponding formal gates on the decomposition.

Since the decomposed Lifeline is interpreted as an InteractionUse, the semantics of a PartDecomposition is the semantics of the Interaction referenced by the decomposition where the gates and parameters have been matched.

That a CombinedFragment is extra-global depicts that there is a CombinedFragment with the same operator covering the decomposed Lifeline in its Interaction. The full understanding of that (higher level) CombinedFragment must be acquired through combining the operands of the decompositions operand by operand.

Notation

PartDecomposition is designated by a referencing clause in the head of the Lifeline as can be seen in the notation section of "Lifeline (from BasicInteractions, Fragments)" on page 475. See also Figure 14.20.

If the part decomposition is denoted inline under the decomposed lifeline and the decomposition clause is the keyword "strict," this indicates that the constructs on all sub lifelines within the inline decomposition are ordered in strict sequence (see "CombinedFragment (from Fragments)" on page 453 on the "strict" operator).

Extraglobal CombinedFragments have their rectangular frame go outside the boundaries of the decomposition Interaction.

Style Guidelines

The name of an Interaction that is involved in decomposition would benefit from including in the name, the name of the type of the Part being decomposed and the name of the Interaction originating the decomposition. This is shown in Figure 14.20 where the decomposition is called *AC_UserAccess* where 'AC' refers to *ACSystem*, which is the type of the Lifeline and *UserAccess* is the name of the Interaction where the decomposed lifeline is contained.

Examples

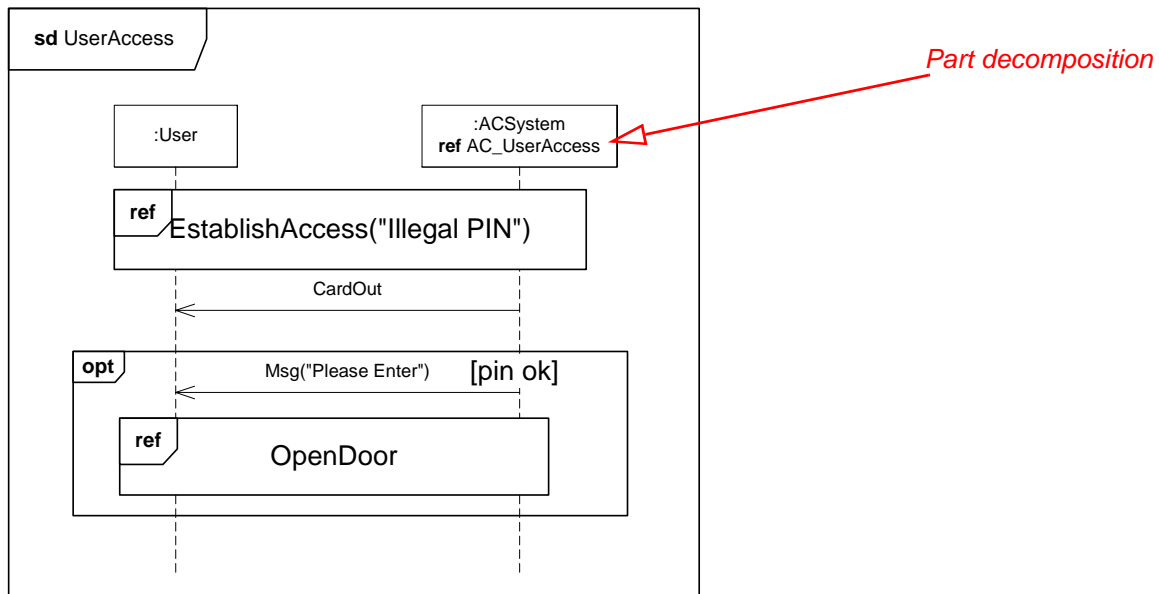


Figure 14.20 - Part Decomposition - the decomposed part

In Figure 14.20 we see how *ACSystem* within *UserAccess* is to be decomposed to *AC_UserAccess*, which is an Interaction owned by class *ACSystem*.

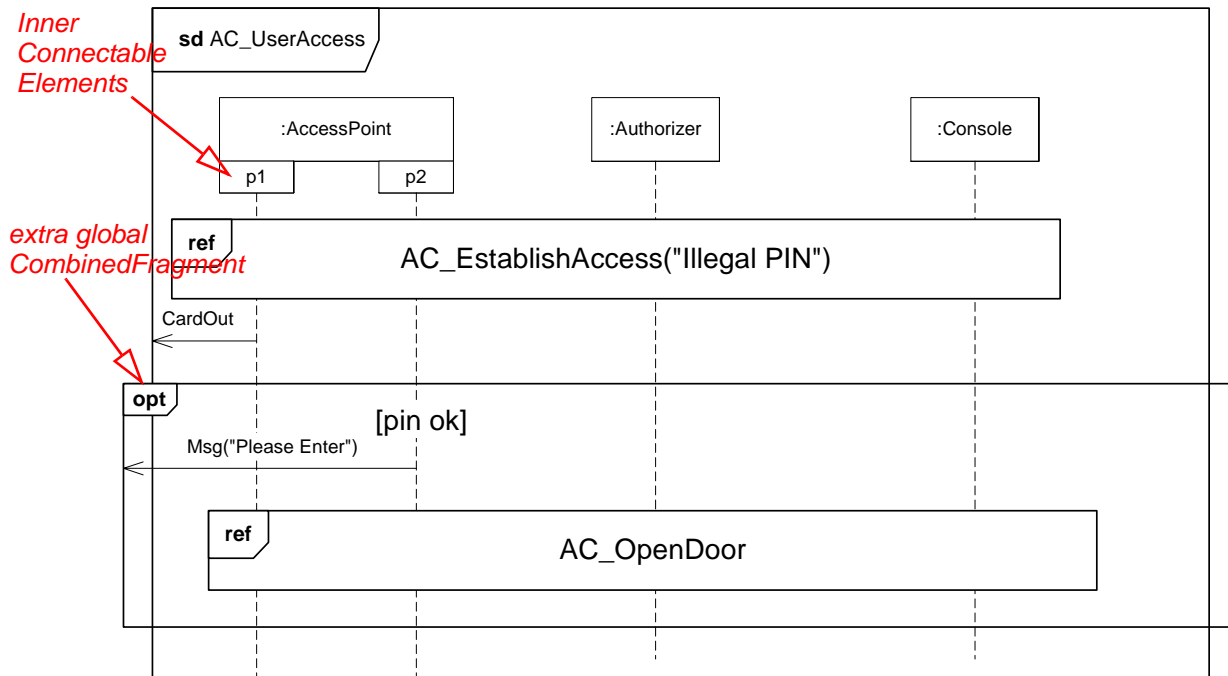


Figure 14.21 - PartDecomposition - the decomposition

In Figure 14.21 we see that *AC_UserAccess* has global constructs that match the constructs of *UserAccess* covering *ACSystem*.

In particular we notice the “extra global interaction group” that goes beyond the frame of the Interaction. This construct corresponds to a *CombinedFragment* of *UserAccess*. However, we want to indicate that the operands of extra global interaction groups are combined one-to-one with similar extra global interaction groups of other decompositions of the same original *CombinedFragment*.

As a notational shorthand, decompositions can also be shown “inline.” In Figure 14.21 we see that the inner *ConnectableElements* of *:AccessPoint* (*p1* and *p2*) are represented by Lifelines already on this level.

Changes from previous UML

PartDecomposition did not appear in UML 1.x.

14.3.27 SendOperationEvent (from BasicInteractions)

Generalizations

- “MessageEvent (from Communications)” on page 431

Description

A *SendOperationEvent* models the invocation of an operation call.

Attributes

No additional attributes

Associations

- operation : Operation [1] The operation associated with this event.

Constraints

No additional constraints

Semantics

A send operation event specifies the sending of a request to invoke a specific operation on an object. The send operation event may result in the occurrence of a call event in the receiver object (see “Common Behaviors” on page 407), and may consequentially cause the execution of a behavior by the receiver object.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.28 SendSignalEvent (from BasicInteractions)**Generalizations**

- “MessageEvent (from Communications)” on page 431

Description

A SendSignalEvent models the sending of a signal.

Attributes

No additional attributes

Associations

- signal : Signal [1] The signal associated with this event.

Constraints

No additional constraints

Semantics

A send signal event specifies the sending of a message to a receiver object. The send signal event may result in the occurrence of a signal event in the receiver object (see “Common Behaviors” on page 407), and may consequentially cause the execution of a behavior by the receiver object. The sending object will not block waiting for a reply, but will continue its execution immediately.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.29 StateInvariant (from BasicInteractions)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 471

Description

A StateInvariant is a runtime constraint on the participants of the interaction. It may be used to specify a variety of different kinds of constraints, such as values of attributes or variables, internal or external states, and so on.

A StateInvariant is an InteractionFragment and it is placed on a Lifeline.

Associations

- invariant: Constraint[1] A Constraint that should hold at runtime for this StateInvariant.
- covered: Lifeline[1] References the Lifeline on which the StateInvariant appears. Specializes InteractionFragment.covered

Semantics

The Constraint is assumed to be evaluated during runtime. The Constraint is evaluated immediately prior to the execution of the next OccurrenceSpecification such that all actions that are not explicitly modeled have been executed. If the Constraint is true, the trace is a valid trace; if the Constraint is false, the trace is an invalid trace. In other words all traces that have a StateInvariant with a false Constraint are considered invalid.

Notation

The possible associated Constraint is shown as text in curly brackets on the lifeline. See example in Figure 14.24 on page 494.

Presentation Options

A StateInvariant can optionally be shown as a Note associated with an OccurrenceSpecification.

The state symbol represents the equivalent of a constraint that checks the state of the object represented by the Lifeline. This could be the internal state of the classifierBehavior of the corresponding Classifier, or it could be some external state based on a “black-box” view of the Lifeline. In the former case, and if the classifierBehavior is described by a state machine, the name of the state should match the hierarchical name of the corresponding state of the state machine.

The regions represent the orthogonal regions of states. The identifier need only define the state partially. The value of the constraint is true if the specified state information is true.

The example in Figure 14.24 also shows this presentation option.

14.4 Diagrams

Interaction diagrams come in different variants. The most common variant is the Sequence Diagram (“Sequence Diagrams” on page 488) that focuses on the Message interchange between a number of Lifelines. Communication Diagrams (“Communication Diagrams” on page 496) show interactions through an architectural view where the arcs between the communicating Lifelines are decorated with description of the passed Messages and their sequencing. Interaction Overview Diagrams (“Interaction Overview Diagrams” on page 499) are a variant of Activity Diagrams that define interactions in a way that promotes overview of the control flow. In the Annexes one may also find optional diagram notations such as Timing Diagrams and Interaction Tables.

Sequence Diagrams

The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the Message interchange between a number of Lifelines.

A sequence diagram describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding OccurrenceSpecifications on the Lifelines. The Interactions that are described by Sequence Diagrams are described in this chapter.

Graphic Nodes

The graphic nodes that can be included in sequence diagrams are shown in Table 14.1.

Table 14.1 - Graphic nodes included in sequence diagrams

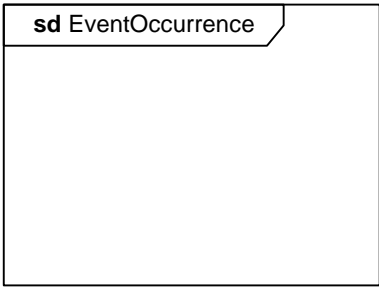
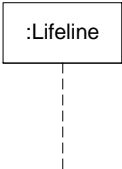
Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 467.
Lifeline		See “Lifeline (from BasicInteractions, Fragments)” on page 475.

Table 14.1 - Graphic nodes included in sequence diagrams

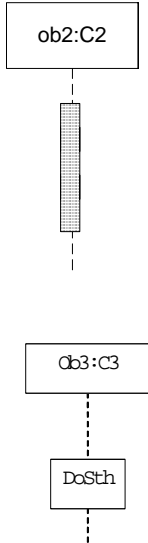

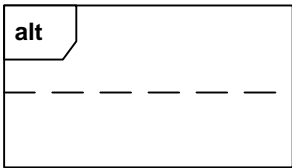
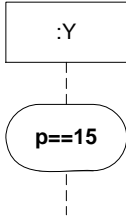
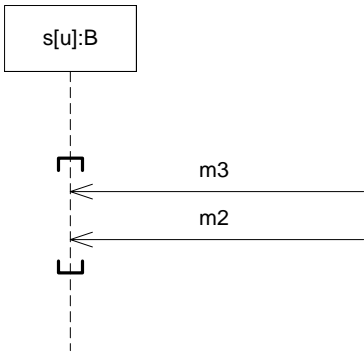

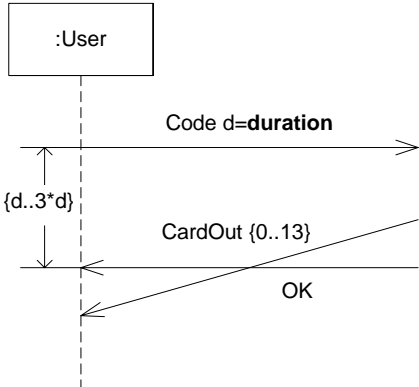
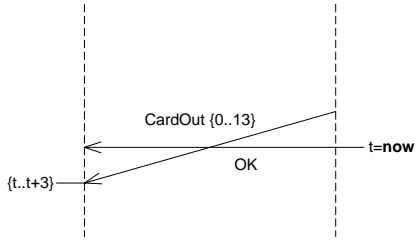
Node Type	Notation	Reference
Execution Specification		See “CombinedFragment (from Fragments)” on page 453. See also “Lifeline (from BasicInteractions, Fragments)” on page 475 and “ExecutionSpecification (from BasicInteractions)” on page 464.
InteractionUse		See “InteractionUse (from Fragments)” on page 473.
CombinedFragment		See “CombinedFragment (from Fragments)” on page 453.
StateInvariant / Continuations		See “Continuation (from Fragments)” on page 459 and “StateInvariant (from BasicInteractions)” on page 487.

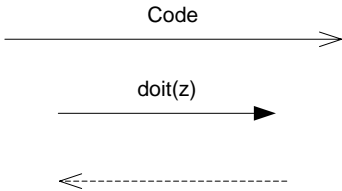
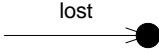
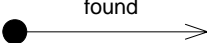

Table 14.1 - Graphic nodes included in sequence diagrams

Node Type	Notation	Reference
Coregion		See explanation under <i>parallel</i> in “CombinedFragment (from Fragments)” on page 453.
Stop		See Figure 14.11 on page 458.
DurationConstraint Duration Observation		See Figure 14.26 on page 496.
TimeConstraint TimeObservation		See Figure 14.26 on page 496.

Graphic Paths

The graphic paths between the graphic nodes are given in Table 14.2

Table 14.2 - Graphic paths included in sequence diagrams

Node Type	Notation	Reference
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. These are all <i>complete</i> messages. See “Message (from BasicInteractions)” on page 477.
Lost Message		Lost messages are messages with known sender, but the reception of the message does not happen. See “Message (from BasicInteractions)” on page 477.
Found Message		Found messages are messages with known receiver, but the sending of the message is not described within the specification. See “Message (from BasicInteractions)” on page 477.
GeneralOrdering		See “GeneralOrdering (from BasicInteractions)” on page 466.