*Package CompleteActivities*

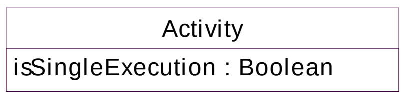


**Figure 12.12 - Elements (CompleteActivities)**

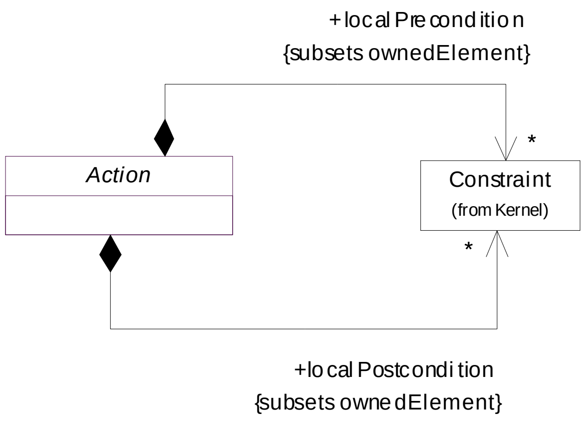


**Figure 12.13 - Constraints (CompleteActivities)**

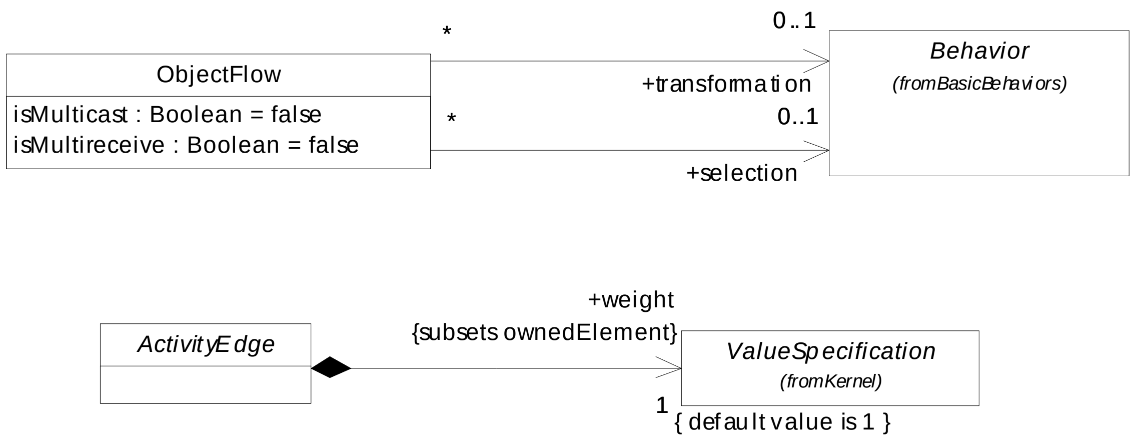


**Figure 12.14 - Flows (CompleteActivities)**

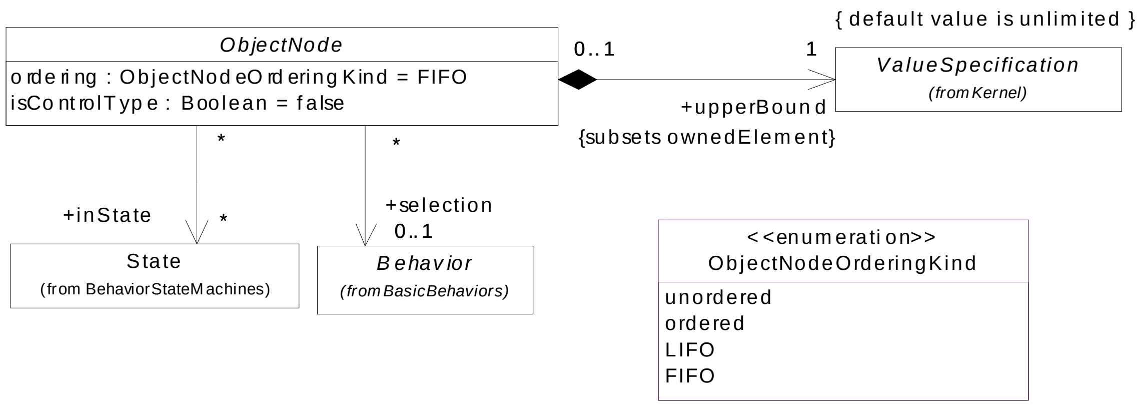


**Figure 12.15 - Object nodes (CompleteActivities)**

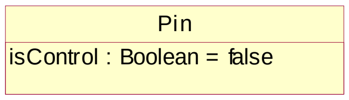


**Figure 12.16 - Control pins**

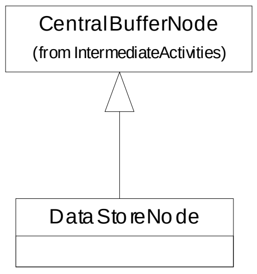


**Figure 12.17 - Data stores**

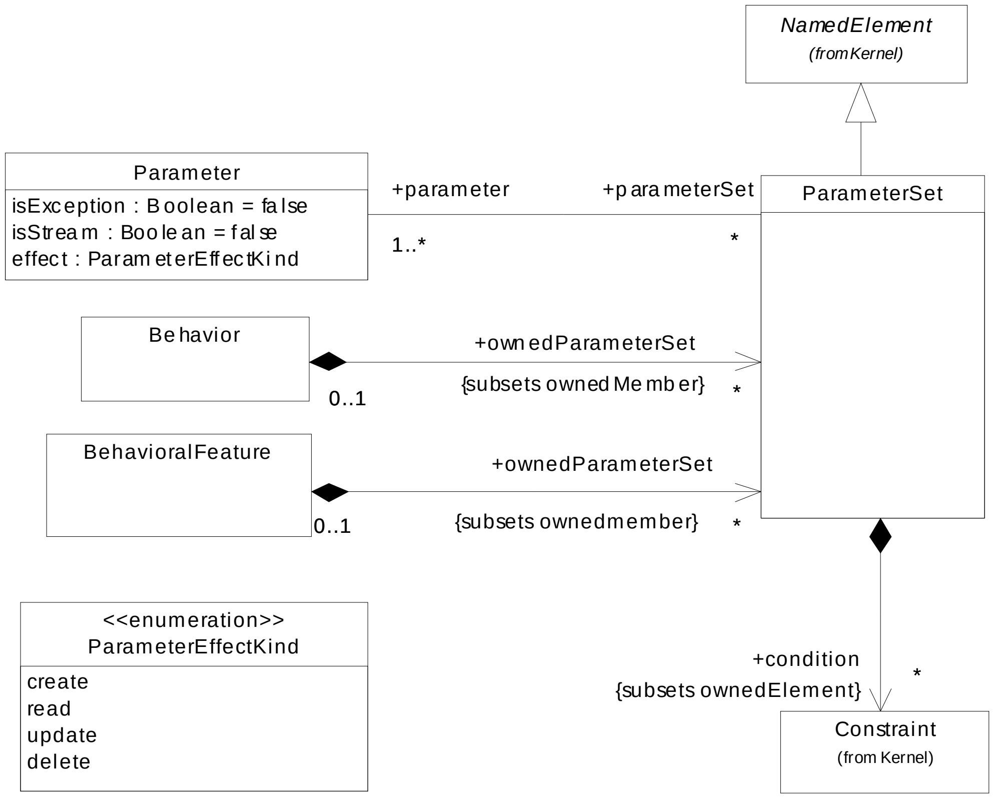


**Figure 12.18 Parameter sets**

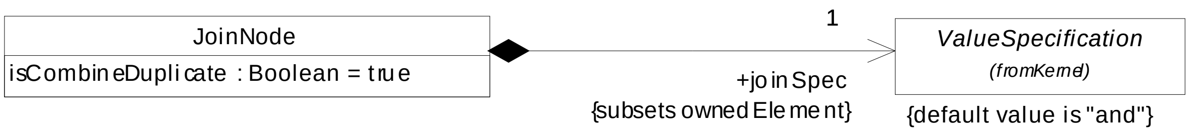


**Figure 12.19 - Control nodes (CompleteActivities)**

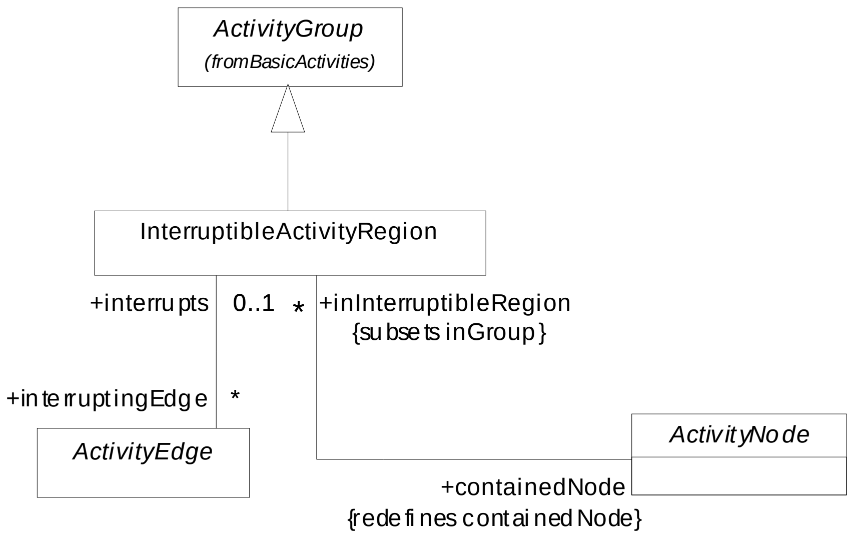
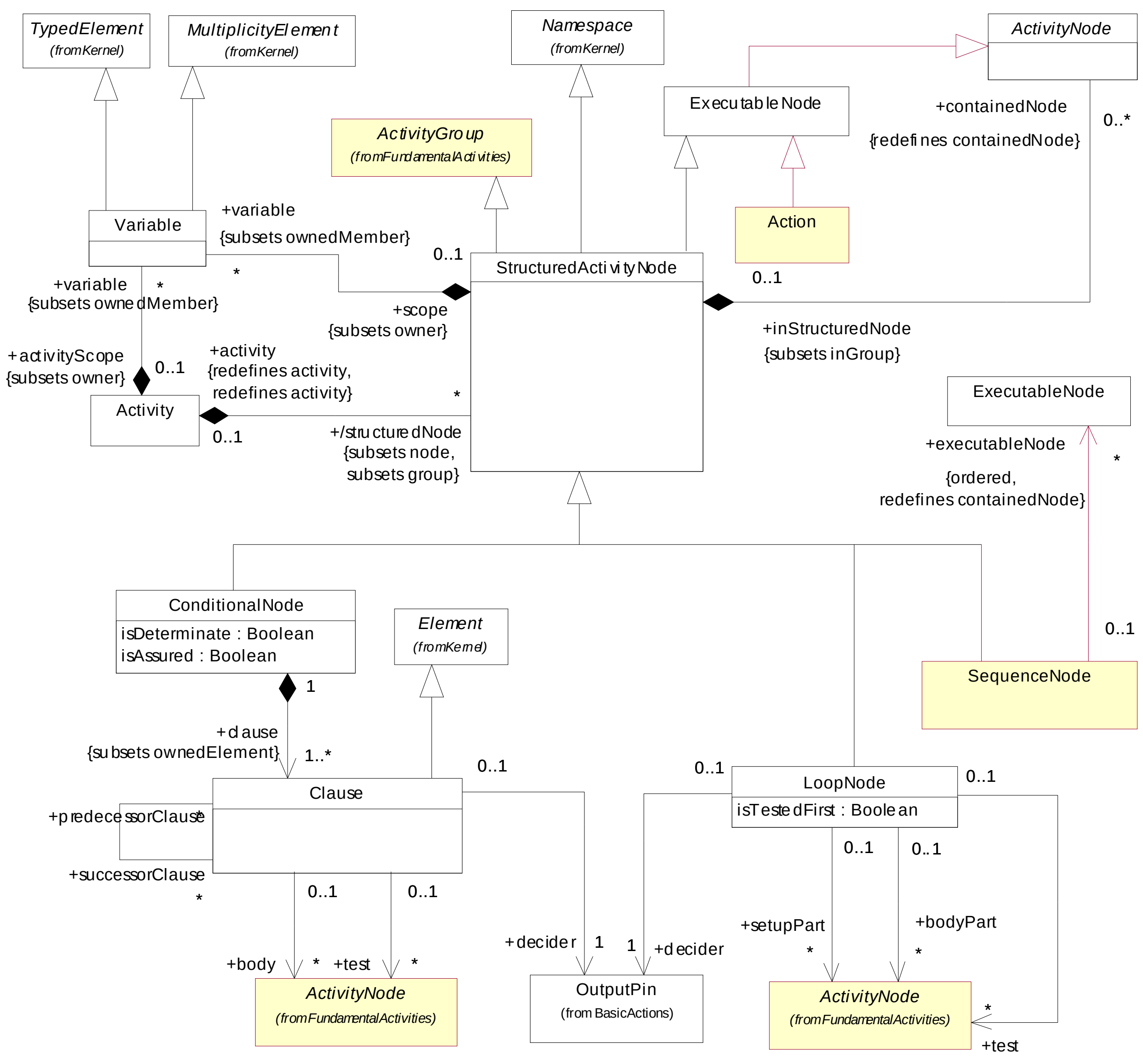


**Figure 12.20 - Interruptible regions**

## *Package StructuredActivities*



TypedElement (fromKernel)
MultiplicityElement (fromKernel)
Namespace (fromKernel)
ActivityNode
ExecutableNode
ActivityGroup (fromFundamentalActivities)
Action
Variable
+variable {subsets ownedMember}
0..1
StructuredActivityNode
+containedNode {redefines containedNode}
0..*
+variable * {subsets ownedMember}
*
+scope {subsets owner}
0..1
+inStructuredNode {subsets inGroup}
+activityScope {subsets owner}
0..1
+activity {redefines activity, redefines activity}
*
Activity
0..1
+/structuredNode {subsets node, subsets group}
ExecutableNode
+executableNode
*
{ordered, redefines containedNode}
ConditionalNode
isDeterminate : Boolean
isAssured : Boolean
Element (fromKernel)
0..1
SequenceNode
1
+clause {subsets ownedElement}
1..*
0..1
0..1
0..1
Clause
LoopNode
isTestedFirst : Boolean
+predecessorClause *
+successorClause *
0..1
0..1
0..1
0..1
+decider 1
1 +decider
+setupPart *
+bodyPart *
+body * +test *
ActivityNode (fromFundamentalActivities)
OutputPin (from BasicActions)
ActivityNode (fromFundamentalActivities)
*
+test

**Figure 12.21 - Structured nodes**

*Package CompleteStructuredActivities*

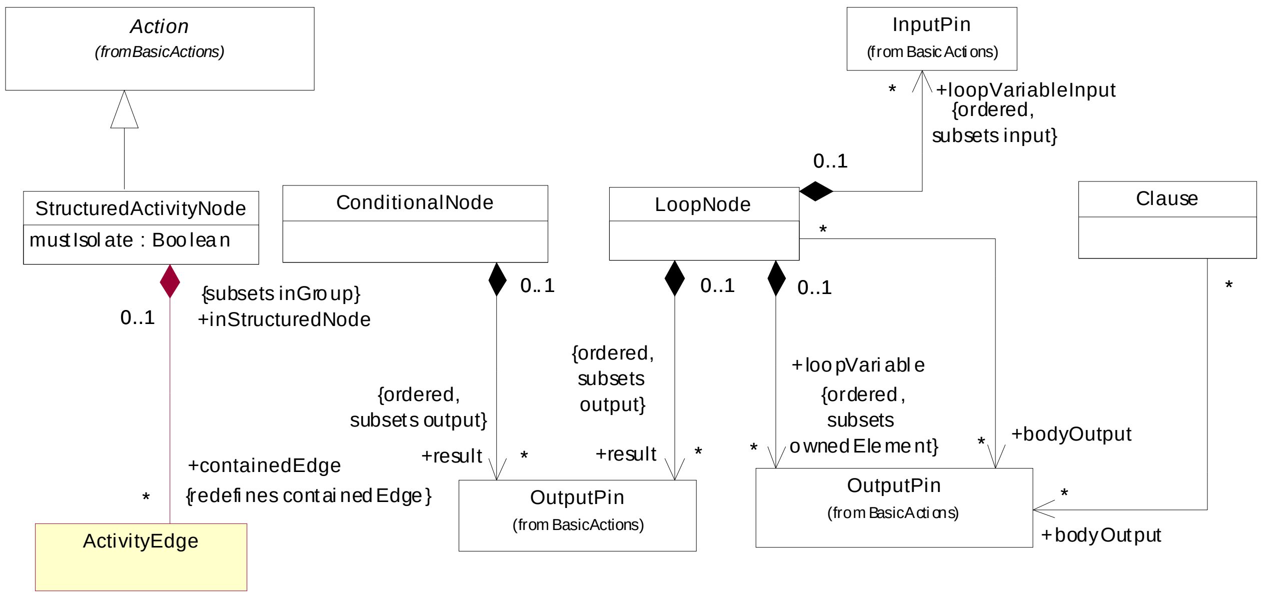


**Figure 12.22 - Structured nodes (CompleteStructuredActivities)**

*Package ExtraStructuredActivities*

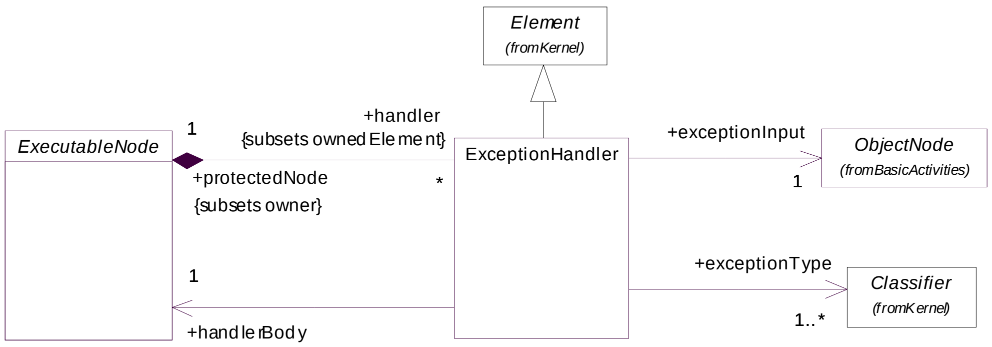


**Figure 12.23 - Exceptions**

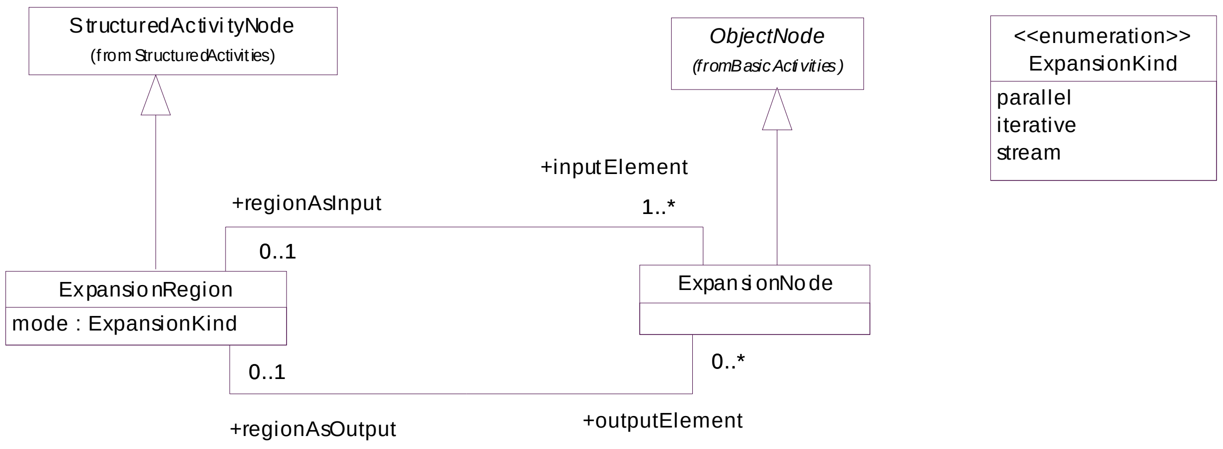


**Figure 12.24 - Expansion regions**

## 12.3 Class Descriptions

### 12.3.1 AcceptEventAction (as specialized)

See "AcceptEventAction (from CompleteActions)" on page 228.

#### Attributes

No additional attributes

#### Associations

No additional associations

#### Constraints

No additional constraints

#### Semantics

If an AcceptEventAction has no incoming edges, then the action starts when the containing activity or structured node does, whichever most immediately contains the action. In addition, an AcceptEventAction with no incoming edges remains enabled after it accepts an event. It does not terminate after accepting an event and outputting a value, but continues to wait for other events. This semantic is an exception to the normal execution rules in Activities. An AcceptEventAction with no incoming edges and contained by a structured node is terminated when its container is terminated.

#### Notation

See "AcceptEventAction (from CompleteActions)" on page 228.

### Examples

Figure 12.25 is an example of the acceptance of a signal indicating the cancellation of an order. The acceptance of the signal causes an invocation of a cancellation behavior. This action is enabled on entry to the activity containing it, therefore no input arrow is shown.

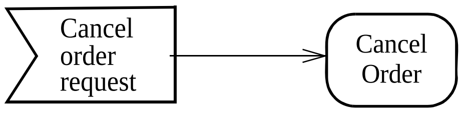


**Figure 12.25 - Accept signal, top level in scope.**

In Figure 12.26, a request payment signal is sent after an order is processed. The activity then waits to receive a payment confirmed signal. Acceptance of the payment confirmed signal is enabled only after the request for payment is sent; no confirmation is accepted until then. When the confirmation is received, the order is shipped.

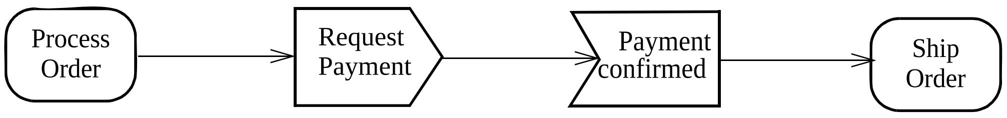


**Figure 12.26 - Accept signal, explicit enable**

In Figure 12.27, the end-of-month accept time event action generates an output at the end of the month. Since there are no incoming edges to the time event action, it is enabled as long as its containing activity or structured node is. It will generate an output at the end of every month.

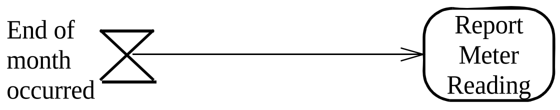


**Figure 12.27 - Repetitive time event**

### Rationale

See "AcceptEventAction (from CompleteActions)" on page 228.

### Changes from previous UML

See "AcceptEventAction (from CompleteActions)" on page 228.

## 12.3.2 Action (from CompleteActivities, FundamentalActivities, StructuredActivities)

### Generalizations

- "ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, StructuredActivities)" on page 323.
- "ExecutableNode (from ExtraStructuredActivities, StructuredActivities)" on page 354.
- "Action (from BasicActions)" on page 230 *(merge increment)*.

### Description

An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action.

*Package CompleteActivities*

In CompleteActivities, action is extended to have pre- and postconditions.

### Attributes

No additional attributes

### Associations

*Package CompleteActivities*

| | |
|---|---|
| • localPrecondition : Constraint [0..*] | Constraint that must be satisfied when execution is started. |
| • localPostcondition : Constraint [0..*] | Constraint that must be satisfied when execution is completed. |

### Constraints

No additional constraints

### Operations

[1] activity operates on Action. It returns the activity containing the action.

```
activity() : Activity;
activity = if self.Activity->size() > 0 then self.Activity else self.group.activity() endif
```

### Semantics

The sequencing of actions are controlled by control edges and object flow edges within activities, which carry control and object tokens respectively (see Activity). Alternatively, the sequencing of actions is controlled by structured nodes, or by a combination of structured nodes and edges. Except where noted, an action can only begin execution when all incoming control edges have tokens, and all input pins have object tokens. The action begins execution by taking tokens from its incoming control edges and input pins. When the execution of an action is complete, it offers tokens in its outgoing control edges and output pins, where they are accessible to other actions.

The steps of executing an action with control and data flow are as follows:

[1] An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). Exceptions to this are listed below. The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others.

[2] An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution. If multiple control tokens are available on a single edge, they are all consumed.

[3] An action continues executing until it has completed. Most actions operate only on their inputs. Some give access to a wider context, such as variables in the containing structured activity node, or the self object, which is the object owning the activity containing the executing action. The detailed semantic of execution an action and definition of completion depends on the particular subclass of action.

[4] When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork), and it terminates. Exceptions to this are listed below. The output tokens are now available to satisfy the control or object flow prerequisites for other action executions.

[5] After an action execution has terminated, its resources may be reclaimed by an implementation, but the details of resource management are not part of this specification and are properly part of an implementation profile.

See ValuePin and Parameter for exceptions to rule for starting action execution.

If a behavior is not reentrant, then no more than one execution of it will exist at any given time. An invocation of a non-reentrant behavior does not start the behavior when the behavior is already executing. In this case, tokens control tokens are discarded, and data tokens collect at the input pins of the invocation action, if their upper bound is greater than one, or upstream otherwise. An invocation of a reentrant behavior will start a new execution of the behavior with newly arrived tokens, even if the behavior is already executing from tokens arriving at the invocation earlier.

### *Package ExtraStructuredActivities*

If an exception occurs during the execution of an action, the execution of the action is abandoned and no regular output is generated by this action. If the action has an exception handler, it receives the exception object as a token. If the action has no exception handler, the exception propagates to the enclosing node and so on until it is caught by one of them. If an exception propagates out of a nested node (action, structured activity node, or activity), all tokens in the nested node are terminated. The data describing an exception is represented as an object of any class.

### *Package CompleteActivities*

Streaming allows an action execution to take inputs and provide outputs while it is executing. During one execution, the action may consume multiple tokens on each streaming input and produce multiple tokens on each streaming output. See Parameter.

Local preconditions and postconditions are constraints that should hold when the execution starts and completes, respectively. They hold only at the point in the flow that they are specified, not globally for other invocations of the behavior at other places in the flow or on other diagrams. Compare to pre and postconditions on Behavior (in Activities). See semantic variations below for their effect on flow.

### Semantic Variation Points

*Package CompleteActivities*

How local pre- and postconditions are enforced is determined by the implementation. For example, violations may be detected at compile time or runtime. The effect may be an error that stops the execution or just a warning, and so on. Since local pre and postconditions are modeler-defined constraints, violations do not mean that the semantics of the invocation is undefined as far as UML goes. They only mean the model or execution trace does not conform to the modeler's intention (although in most cases this indicates a serious modeling error that calls into question the validity of the model).

See variations in ActivityEdge and ObjectNode.

### Notation

Use of action and activity notation is optional. A textual notation may be used instead.

Actions are notated as round-cornered rectangles. The name of the action or other description of it may appear in the symbol. See children of action for refinements.

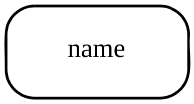


**Figure 12.28 - Action**

*Package CompleteActivities*

Local pre- and postconditions are shown as notes attached to the invocation with the keywords «localPrecondition» and «localPostcondition», respectively.

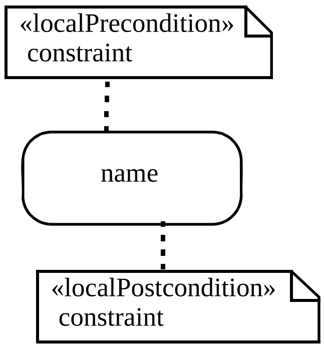


**Figure 12.29 - Local pre- and postconditions**

## Examples

Examples of actions are illustrated below. These perform behaviors called Send Payment and Accept Payment.

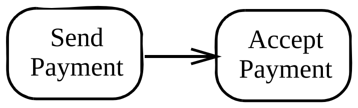


**Figure 12.30 - Examples of actions**

Below is an example of an action expressed in an application-dependent action language:

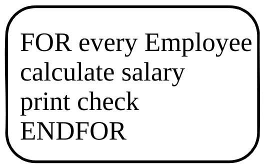


**Figure 12.31 - Example of action with tool-dependent action language**

### *Package CompleteActivities*

The example below illustrates local pre- and postconditions for the action of a drink-dispensing machine. This is considered "local" because a drink-dispensing machine is constrained to operate under these conditions for this particular action. For a machine technician scenario, the situation would be different. Here, a machine technician would have a key to open up the machine, and therefore no money need be inserted to dispense the drink, nor change need be given. In such a situation, the global pre- and postconditions would be all that is required. (Global conditions are described in Activity specification, in the next subsection.) For example, a global precondition for a Dispense Drink activity could be "A drink is selected that the vending machine dispenses." The postcondition, then, would be "The vending machine dispensed the drink that was selected." In other words, there is no global requirement for money and correct change.

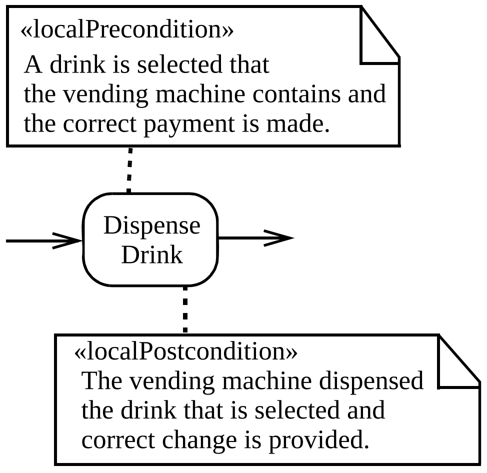

«localPrecondition»
A drink is selected that
the vending machine contains and
the correct payment is made.
Dispense Drink
«localPostcondition»
The vending machine dispensed
the drink that is selected and
correct change is provided.

**Figure 12.32 - Example of an action with local pre/postconditions**

### Rationale

An action represents a single step within an activity, that is, one that is not further decomposed within the activity. An activity represents a behavior that is composed of individual elements that are actions. Note, however, that a call behavior action may reference an activity definition, in which case the execution of the call action involves the execution of the referenced activity and its actions. Similarly for all the invocation actions. An action is therefore simple from the point of view of the activity containing it, but may be complex in its effect and not be atomic. As a piece of structure within an activity model, it is a single discrete element; as a specification of behavior to be performed, it may invoke referenced behavior that is arbitrarily complex. As a consequence, an activity defines a behavior that can be reused in many places, whereas an instance of an action is only used once at a particular point in an activity.

### Changes from previous UML

Explicitly modeled actions as part of activities are new in UML 2.0, and replace ActionState, CallState, and SubactivityState in UML 1.5. They represent a merger of activity graphs from UML 1.5 and actions from UML 1.5.

Local pre and postconditions are new to UML 2.0.

## 12.3.3 ActionInputPin (as specialized)

See “ActionInputPin (from StructuredActions)” on page 231.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

See “ActionInputPin (from StructuredActions)” on page 231.

### Notation

An action input pin with a ReadVariableAction as a fromAction is notated as an input pin with the variable name written beside it. An action input pin with a ReadSelfObject as a fromAction is notated as an input pin with the word “self” written beside it. An action input pin with a ValueSpecification as a fromAction is notated as an input pin with the value specification written beside it.

### Examples

See “ActionInputPin (from StructuredActions)” on page 231.

### 12.3.4 Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)

An activity is the specification of parameterized behavior as the coordinated sequencing of subordinate units whose individual elements are actions. There are actions that invoke activities (directly by “CallBehaviorAction (from BasicActions)” on page 237 or indirectly as methods by “CallOperationAction (from BasicActions)” on page 239).

#### Generalizations

- “Behavior (from BasicBehaviors)” on page 416

#### Description

An activity specifies the coordination of executions of subordinate behaviors, using a control and data flow model. The subordinate behaviors coordinated by these models may be initiated because other behaviors in the model finish executing, because objects and data become available, or because events occur external to the flow. The flow of execution is modeled as activity nodes connected by activity edges. A node can be the execution of a subordinate behavior, such as an arithmetic computation, a call to an operation, or manipulation of object contents. Activity nodes also include flow-of-control constructs, such as synchronization, decision, and concurrency control. Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions. In an object-oriented model, activities are usually invoked indirectly as methods bound to operations that are directly invoked.

Activities may describe procedural computation. In this context, they are the methods corresponding to operations on classes. Activities may be applied to organizational modeling for business process engineering and workflow. In this context, events often originate from inside the system, such as the finishing of a task, but also from outside the system, such as a customer call. Activities can also be used for information system modeling to specify system level processes.

Activities may contain actions of various kinds:

- Occurrences of primitive functions, such as arithmetic functions.
- Invocations of behavior, such as activities.
- Communication actions, such as sending of signals.
- Manipulations of objects, such as reading or writing attributes or associations.

Actions have no further decomposition in the activity containing them. However, the execution of a single action may induce the execution of many other actions. For example, a call action invokes an operation that is implemented by an activity containing actions that execute before the call action completes.

Most of the constructs in the Activity chapter deal with various mechanisms for sequencing the flow of control and data among the actions:

- Object flows for sequencing data produced by one node that is used by other nodes.
- Control flows for sequencing the execution of nodes.
- Control nodes to structure control and object flow. These include decisions and merges to model contingency. These also include initial and final nodes for starting and ending flows. In IntermediateActivities, they include forks and joins for creating and synchronizing concurrent subexecutions.
- Activity generalization to replace nodes and edges.
- Object nodes to represent objects and data as they flow in and out of invoked behaviors, or to represent collections of

tokens waiting to move downstream.

*Package StructuredActivities*

- Composite nodes to represent structured flow-of-control constructs, such as loops and conditionals.

*Package IntermediateActivities*

- Partitions to organize lower-level activities according to various criteria, such as the real-world organization responsible for their performance.

*Package CompleteActivities*

- Interruptible regions and exceptions to represent deviations from the normal, mainline flow of control.

### Attributes

*Package BasicActivities*

- isReadOnly : Boolean = false If *true*, this activity must not make any changes to variables outside the activity or to objects. (This is an assertion, not an executable property. It may be used by an execution engine to optimize model execution. If the assertion is violated by the action, then the model is ill formed.) The default is false (an activity may make non-local changes).

*Package CompleteActivities*

- isSingleExecution : Boolean = false If *true*, all invocations of the activity are handled by the same execution.

### Associations

*Package FundamentalActivities*

- group : ActivityGroup [0..*] Top-level groups in the activity.
- node : ActivityNode [0..*] Nodes coordinated by the activity.

*Package BasicActivities*

- edge : ActivityEdge [0..*] Edges expressing flow between nodes of the activity.

*Package IntermediateActivities*

- partition : ActivityPartition [0..*] Top-level partitions in the activity.

*Package StructuredActivities*

- /structuredNode : StructuredActivityNode [0..*] Top-level structured nodes in the activity. Subsets
- variable : Variable [0..*] Top-level variables in the activity. Subsets *Namespace::ownedMember.*

### Constraints

[1] The nodes of the activity must include one ActivityParameterNode for each parameter.

[2] An activity cannot be autonomous and have a classifier or behavioral feature context at the same time.

## Semantics

The semantics of activities is based on token flow. By *flow*, we mean that the execution of one node affects, and is affected by, the execution of other nodes, and such dependencies are represented by *edges* in the activity diagram. A *token* contains an object, datum, or locus of control, and is present in the activity diagram at a particular node. Each token is distinct from any other, even if it contains the same value as another. A node may begin execution when specified conditions on its input tokens are satisfied; the conditions depend on the kind of node. When a node begins execution, tokens are accepted from some or all of its input edges and a token is placed on the node. When a node completes execution, a token is removed from the node and tokens are offered to some or all of its output edges. See later in this section for more about how tokens are managed.

All restrictions on the relative execution order of two or more actions are explicitly constrained by flow relationships. If two actions are not directly or indirectly ordered by flow relationships, they may execute concurrently. This does not require parallel execution; a specific execution engine may choose to perform the executions sequentially or in parallel, as long as any explicit ordering constraints are satisfied. In most cases, there are some flow relationships that constrain execution order. Concurrency is supported in IntermediateActivities, but not in BasicActivities.

Activities can be parameterized, which is a capability inherited from Behavior (see 12.3.9, "ActivityParameterNode (from BasicActivities)," on page 326). Functionality inherited from Behavior also supports the use of activities on classifiers and as methods for behavioral features. The classifier, if any, is referred to as the *context* of the activity. At runtime, the activity has access to the attributes and operations of its context object and any objects linked to the context object, transitively. An activity that is also a method of a behavioral feature has access to the parameters of the behavioral feature. In workflow terminology, the scope of information an activity uses is called the process-relevant data. Implementations that have access to metadata can define parameters that accept entire activities or other parts of the user model.

An activity with a classifier context, but that is not a method of a behavioral feature, is invoked when the classifier is instantiated. An activity that is a method of a behavioral feature is invoked when the behavioral feature is invoked. The Behavior metaclass also provides parameters, which must be compatible with the behavioral feature it is a method of, if any. Behavior also supports overriding of activities used as inherited methods. See the Behavior metaclass for more information.

Activities can also be invoked directly by other activities rather than through the call of a behavioral feature that has an activity as a method. This functional or monomorphic style of invocation is useful at the stage of development where focus is on the activities to be completed and goals to be achieved. Classifiers responsible for each activity can be assigned at a later stage by declaring behavioral features on classifiers and assigning activities as methods for these features. For example, in business reengineering, an activity flow can be optimized independently of which departments or positions are later assigned to handle each step. This is why activities are autonomous when they are not assigned to a classifier.

Regardless of whether an activity is invoked through a behavioral feature or directly, inputs to the invoked activity are supplied by an invocation action in the calling activity, which gets its inputs from incoming edges. Likewise an activity invoked from another activity produces outputs that are delivered to an invocation action, which passes them onto its outgoing edges.

An activity execution represents an execution of the activity. An activity execution, as a reflective object, can support operations for managing execution, such as starting, stopping, aborting, and so on; attributes, such as how long the process has been executing or how much it costs; and links to objects, such as the performer of the execution, who to report completion to, or resources being used, and states of execution such as started, suspended, and so on. Used this way activity is the modeling basis for the WfProcess interface in the OMG Workflow Management Facility, www.omg.org/cgi-bin/doc?formal/00-05-02. It is expected that profiles will include class libraries with standard classes that are used as root classes for activities in the user model. Vendors may define their own libraries, or support user-defined features on activity classes.

Nodes and edges have token flow rules. Nodes control when tokens enter or leave them. Edges have rules about when a token may be taken from the source node and moved to the target node. A token traverses an edge when it satisfies the rules for target node, edge, and source node all at once. This means a source node can only offer tokens to the outgoing edges, rather than force them along the edge, because the tokens may be rejected by the edge or the target node on the other side. Multiple tokens offered to an edge at once is the same as if they were offered one at a time. Since multiple edges can leave the same node, token flow semantics is highly distributed and subject to timing issues and race conditions, as is any distributed system. There is no specification of the order in which rules are applied on the various nodes and edges in an activity. It is the responsibility of the modeler to ensure that timing issues do not affect system goals, or that they are eliminated from the model. Execution profiles may tighten the rules to enforce various kinds of execution semantics. Start at ActivityEdge and ActivityNode to see the token management rules.

Tokens cannot “rest” at *control nodes*, such as decisions and merges, waiting to move downstream. Control nodes act as traffic switches managing tokens as they make their way between object nodes and actions, which are the nodes where tokens can rest for a period of time. Initial nodes are excepted from this rule.

A data token with no value in is called the *null* token. It can be passed along and used like any other token. For example, an action can output a null token and a downstream decision point can test for it and branch accordingly. Null tokens satisfy the type of all object nodes.

The semantics of activities is specified in terms of these token rules, but only for the purpose of describing the expected runtime behavior. Token semantics is not intended to dictate the way activities are implemented, despite the use of the term “execution.” They only define the sequence and conditions for behaviors to start and stop. Token rules may be optimized in particular cases as long as the effect is the same.

#### *Package IntermediateActivities*

Activities can have multiple tokens flowing in them at any one time, if required. Special nodes called *object nodes* provide and accept objects and data as they flow in and out of invoked behaviors, and may act as buffers, collecting tokens as they wait to move downstream.

#### *Package CompleteActivities*

Each time an activity is invoked, the isSingleExecution attribute indicates whether the same execution of the activity handles tokens for all invocations, or a separate execution of the activity is created for each invocation. For example, an activity that models a manufacturing plant might have a parameter for an order to fill. Each time the activity is invoked, a new order enters the flow. Since there is only one plant, one execution of the activity handles all orders. This applies even if the behavior is a method, for example, on each order. If a single execution of the activity is used for all invocations, the modeler must consider the interactions between the multiple streams of tokens moving through the nodes and edges. Tokens may reach bottlenecks waiting for other tokens ahead of them to move downstream, they may overtake each other due to variations in the execution time of invoked behaviors, and most importantly, may abort each other with constructs such as activity final.

If a separate execution of the activity is used for each invocation, tokens from the various invocations do not interact. For example, an activity that is the behavior of a classifier, is invoked when the classifier is instantiated, and the modeler will usually want a separate execution of the activity for each instance of the classifier. The same is true for modeling methods in common programming languages, which have separate stack frames for each method call. A new activity execution for each invocation reduces token interaction, but might not eliminate it. For example, an activity may have a loop creating tokens to be handled by the rest of the activity, or an unsynchronized flow that is aborted by an activity final. In these cases, modelers must consider the same token interaction issues as using a single activity execution for all invocations. Also see the effect of non-reentrant behaviors described at Except in CompleteActivities, each invocation of an activity is executed separately; tokens from different invocations do not interact.

Nodes and edges inherited from more general activities can be replaced. See RedefinableElement for more information on overriding inherited elements.

#### *Package IntermediateActivities*

If a single execution of the activity is used for all invocations, the modeler must consider additional interactions between tokens. Tokens may reach bottlenecks waiting for tokens ahead of them to move downstream, they may overtake each other due to the ordering algorithm used in object node buffers, or due to variations in the execution time of invoked behaviors, and most importantly, may abort each other with constructs such as activity final, exception outputs, and interruptible regions.

#### *Package CompleteActivities*

Complete activities add functionality that also increases interaction. For example, streaming outputs create tokens to be handled by the rest of the activity. In these cases, modelers must consider the same token interaction issues even when using a separate execution of activity execution for all invocations.

Interruptible activity regions are groups of nodes within which all execution can be terminated if an interruptible activity edge is traversed leaving the region.

See “ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, StructuredActivities)” and “ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)” for more information on the way activities function. An activity with no nodes and edges is well-formed, but unspecified. It may be used as an alternative to a generic behavior in activity modeling. See “ActivityPartition (from IntermediateActivities)” for more information on grouping mechanisms in activities.

### Semantic Variation Points

No specific variations in token management are defined, but extensions may add new types of tokens that have their own flow rules. For example, a BPEL extension might define a failure token that flows along edges that reject other tokens. Or an extension for systems engineering might define a new control token that terminates executing actions.

### Notation

Use of action and activity notation is optional. A textual notation may be used instead.

The notation for an activity is a combination of the notations of the nodes and edges it contains, plus a border and name displayed in the upper left corner. Activity parameter nodes are displayed on the border. Actions and flows that are contained in the activity are also depicted.

Pre- and post condition constraints, inherited from Behavior, are shown as with the keywords «precondition» and «postcondition», respectively. These apply globally to all uses of the activity. See Figure 12.33 and Behavior in Common Behavior. Compare to local pre- and postconditions on Action.

(CompleteActivities) The keyword «singleExecution» is used for activities that execute as a single shared execution. Otherwise, each invocation executes in its space. See the notation sections of the various kinds of nodes and edges for more information.

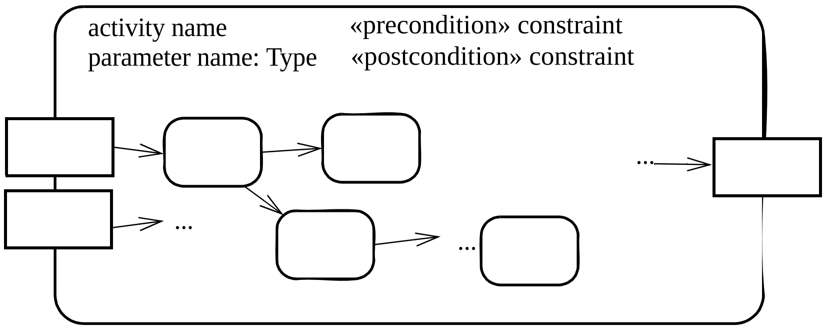


**Figure 12.33 - Activity notation**

The notation for classes can be used for diagramming the features of a reflective activity as shown below, with the keyword “activity” to indicate it is an activity class. Association and state machine notation can also be used as necessary.

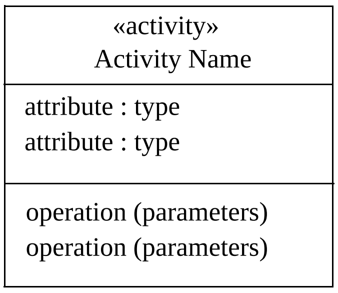


**Figure 12.34 - Activity class notation**

### Presentation Options

The round-cornered border of Figure 12.33 may be replaced with the frame notation described in Annex A. Activity parameter nodes are displayed on the frame. The round-cornered border or frame may be omitted completely. See the presentation option for “ActivityParameterNode (from BasicActivities)” on page 326.

## Examples

The definition of Process Order below uses the border notation to indicate that it is an activity. It has pre- and post conditions on the order (see Behavior). All invocations of it use the same execution.

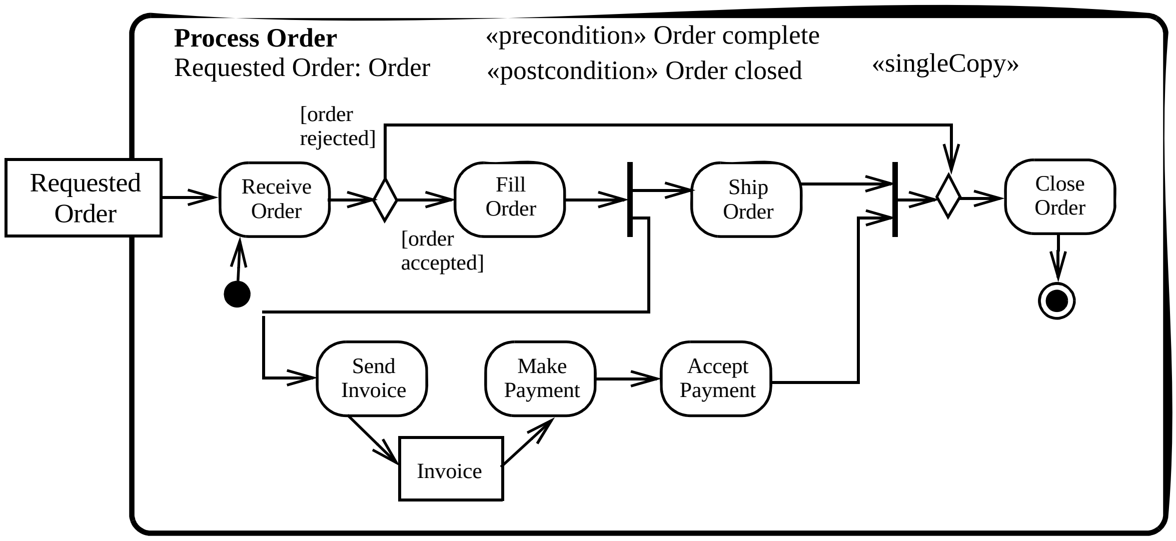


**Figure 12.35 - Example of an activity with input parameter**

The diagram below is based on a standard part selection workflow within an airline design process. Notice that the Standards Engineer insures that the substeps in Provide Required Part are performed in the order specified and under the conditions specified, but doesn't necessarily perform the steps. Some of them are performed by the Design Engineer even though the Standards Engineer is managing the process. The Expert Part Search behavior can result in a part found or not. When a part is not found, it is assigned to the Assign Standards Engineer activity. Lastly, Schedule Part Mod Workflow invocation produces entire activities and they are passed to subsequent invocations for scheduling and execution (i.e., Schedule Part Mod Workflow, Execute Part Mod Workflow, and Research Production Possibility). In other words, behaviors can produce tokens that are activities that can in turn be executed; in short, runtime activity generation and execution.

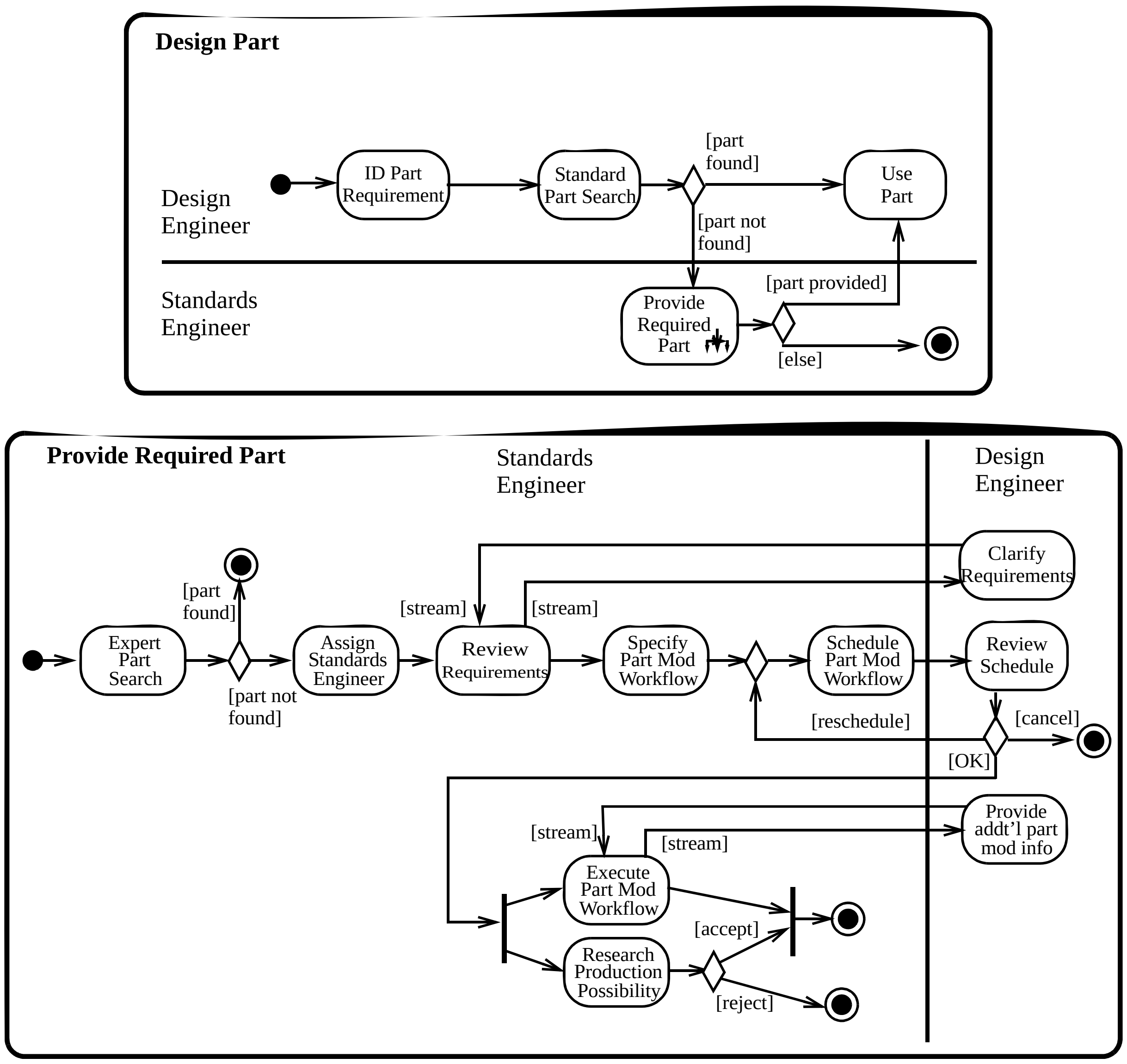


Figure 12.36 - Workflow example

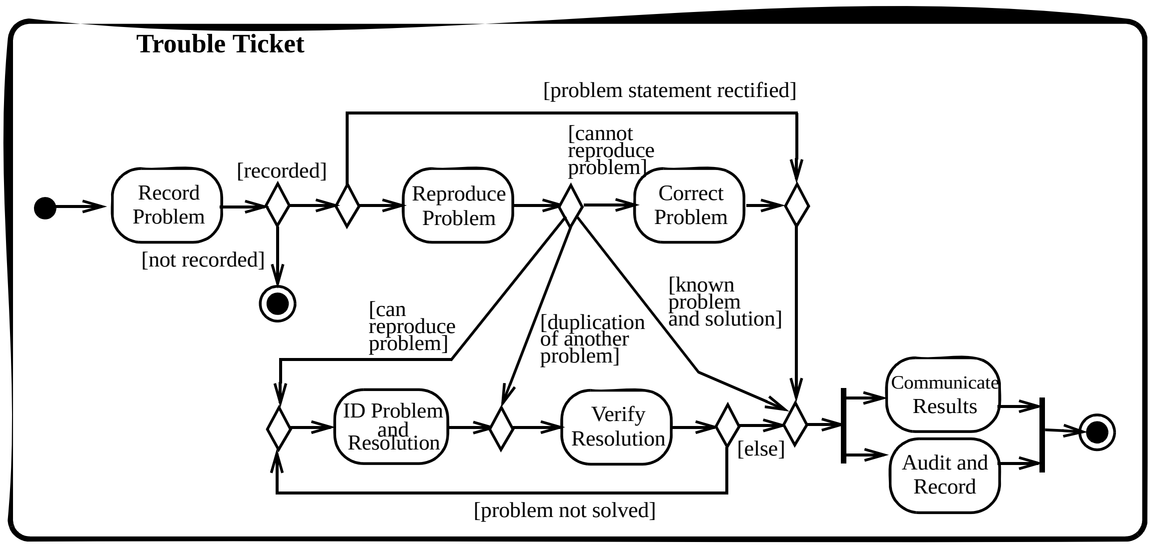


**Figure 12.37 - Workflow example**

Below is an example of using class notation to show the class features of an activity. Associations and state machines can also be shown.

| «activity»<br>Fill Order |
|---|
| costSoFar : USD<br>timeToComplete : Integer |
| suspend ()<br>resume () |

**Figure 12.38 - Activity class with attributes and operations**

## Rationale

Activities are introduced to flow models that coordinate other behaviors, including other flow models. It supports class features to model control and monitoring of executing processes, and relating them to other objects (for example, in an organization model).

## Changes from previous UML

Activity replaces ActivityGraph in UML 1.5. Activities are redesigned to use a Petri-like semantics instead of state machines. Among other benefits, this widens the number of flows that can be modeled, especially those that have parallel flows. Activity also replaces procedures in UML 1.5, as well as the other control and sequencing aspects, including composite and collection actions.

## 12.3.5 ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)

An activity edge is an abstract class for directed connections between two activity nodes.

### Generalizations

- “RedefinableElement (from Kernel)” on page 125

### Description

ActivityEdge is an abstract class for the connections along which tokens flow between activity nodes. It covers control and data flow edges. Activity edges can control token flow.

*Package CompleteActivities*

Complete activity edges can be contained in interruptible regions.

### Attributes

No additional attributes

### Associations

*Package BasicActivities*

| | |
|---|---|
| activity : Activity[0..1] — Activity containing the edge. | |
| /inGroup : ActivityGroup[0..*] | Groups containing the edge. Multiplicity specialized to [0..1] for StructuredActivityGroup. |
| redefinedElement: ActivityEdge [0..*] | Inherited edges replaced by this edge in a specialization of the activity. |
| source ActivityNode [1..1] | Node from which tokens are taken when they traverse the edge. |
| target : ActivityNode [1..1] | Node to which tokens are put when they traverse the edge. |

*Package IntermediateActivities*

| | |
|---|---|
| inPartition : Partition [0..*] | Partitions containing the edge. |
| guard : ValueSpecification [1..1] = true | Specification evaluated at runtime to determine if the edge can be traversed. |

*Package CompleteStructuredActivities*

| | |
|---|---|
| inStructuredNode : StructuredActivityNode [0..1] | Structured activity node containing the edge. |

*Package CompleteActivities*

| | |
|---|---|
| interrupts : InterruptibleActivityRegion [0..1] | Region that the edge can interrupt. |
| weight : ValueSpecification [1..1] = 1 | Number of objects consumed from the source node on each traversal. |

### Constraints

[1] The source and target of an edge must be in the same activity as the edge.

[2] Activity edges may be owned only by activities or groups.

*Package CompleteStructuredActivities*

[1] Activity edges may be owned by at most one structured node.

### Semantics

Activity edges are directed connections, that is, they have a source and a target, along which tokens may flow.

Other rules for when tokens may be passed along the edge depend on the kind of edge and characteristics of its source and target. See the children of ActivityEdge and ActivityNode. The rules may be optimized to a different algorithm as long as the effect is the same.

The guard must evaluate to true for every token that is offered to pass along the edge. Tokens in the intermediate level of activities can only pass along the edge individually at different times. See application of guards at DecisionNode.

*Package CompleteActivities*

Any number of tokens can pass along the edge, in groups at one time, or individually at different times. The weight attribute dictates the minimum number of tokens that must traverse the edge at the same time. It is a value specification evaluated every time a new token becomes available at the source. It must evaluate to a positive LiteralUnlimitedNatural, and may be a constant. When the minimum number of tokens are offered, all the tokens at the source are offered to the target all at once. The guard must evaluate to true for each token. If the guard fails for any of the tokens, and this reduces the number of tokens that can be offered to the target to less than the weight, then all the tokens fail to be offered. An unlimited weight means that all the tokens at the source are offered to the target. This can be combined with a join to take all of the tokens at the source when certain conditions hold. See examples in Figure 12.45. A weaker but simpler alternative to weight is grouping information into larger objects so that a single token carries all necessary data. See additional functionality for guards at DecisionNode.

Other rules for when tokens may be passed along the edge depend on the kind of edge and characteristics of its source and target. See the children of ActivityEdge and ActivityNode. The rules may be optimized to a different algorithm as long as the effect is the same. For example, if the target is an object node that has reached its upper bound, no token can be passed. The implementation can omit unnecessary weight evaluations until the downstream object node can accept tokens.

Edges can be named, by inheritance from RedefinableElement, which is a NamedElement. However, edges are not required to have unique names within an activity. The fact that Activity is a Namespace, inherited through Behavior, does not affect this, because the containment of edges is through ownedElement, the general ownership metaassociation for Element that does not imply unique names, rather than ownedMember.

Edges inherited from more general activities can be replaced. See RedefinableElement for more information on overriding inherited elements.

### Semantic Variation Points

See variations at children of ActivityEdge and ActivityNode.

## Notation

An activity edge is notated by a stick-arrowhead line connecting two activity nodes. If the edge has a name, it is notated near the arrow.

name
Regular activity edge
Activity edge with name

**Figure 12.39 - Activity edge notation**

An activity edge can also be notated using a connector, which is a small circle with the name of the edge in it. This is purely notational. It does not affect the underlying model. The circles and lines involved map to a single activity edge in the model. Every connector with a given label must be paired with exactly one other with the same label on the same activity diagram. One connector must have exactly one incoming edge and the other exactly one outgoing edge, each with the same type of flow, object or control. This assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

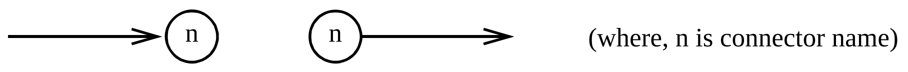


**Figure 12.40 - Activity edge connector notation**

### *Package CompleteActivities*

The weight of the edge may be shown in curly braces that contain the weight. The weight is a value specification that is a positive integer or null, which may be a constant. A weight of null is notated as “all.” When regions have interruptions, a lightning-bolt style activity edge expresses this interruption, see InterruptibleActivityRegion. See Pin for filled arrowhead notation.

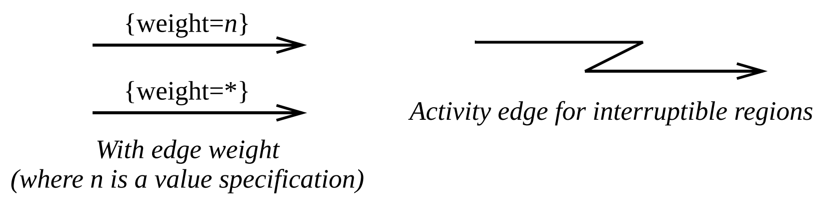

{weight=*n*}
{weight=*}
*With edge weight*
*(where n is a value specification)*
*Activity edge for interruptible regions*

**Figure 12.41 - Activity edge notation**

## Examples

### *Package BasicActivities*

In the example illustrated below, the arrowed line connecting Fill Order to Ship Order is a control flow edge. This means that when the Fill Order behavior is completed, control is passed to the Ship Order. Below it, the same control flow is shown with an edge name. The one at the bottom left employs connectors, instead of a continuous line. On the upper right, the arrowed lines starting from Send Invoice and ending at Make Payment (via the Invoice object node) are object flow edges. This indicates that the flow of Invoice objects goes from Send Invoice to Make Payment.

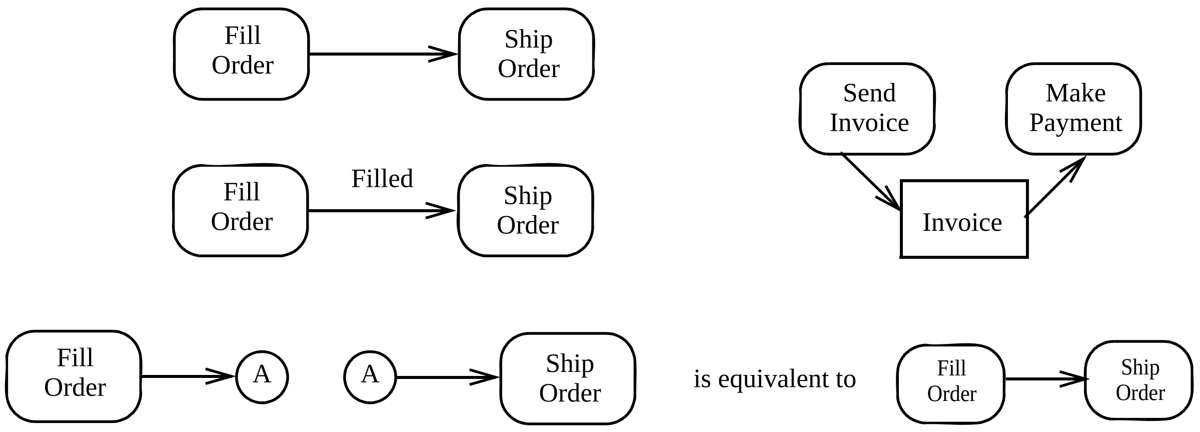


**Figure 12.42 - Activity edge examples**

In the example below, a connector is used to avoid drawing a long edge around one tine of the fork. If a problem is not priority one, the token going to the connector is sent to the merge instead of the one that would arrive from Revise Plan for priority one problems. This is equivalent to the activity shown in Figure 12.44, which is how Figure 12.43 is stored in the model.

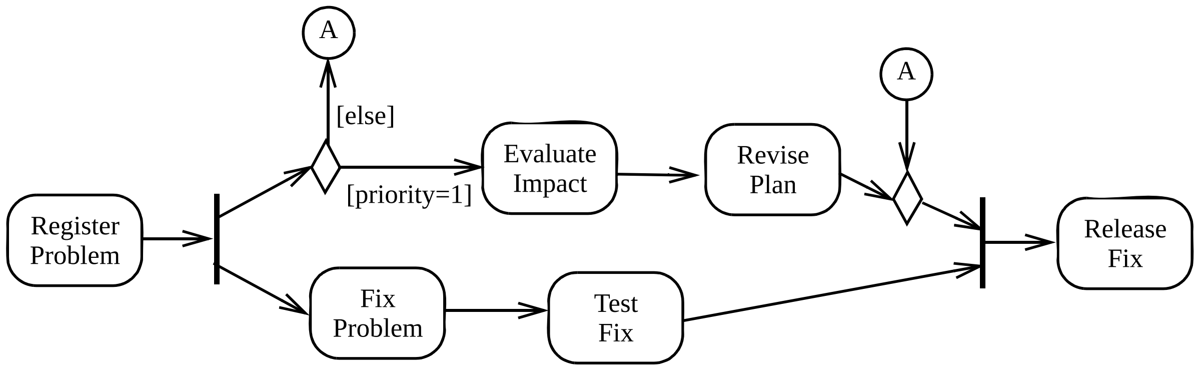


**Figure 12.43 - Connector example**

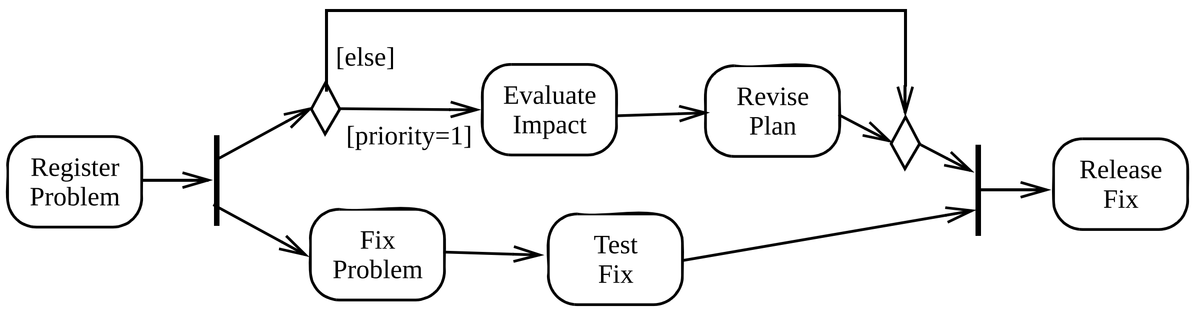


**Figure 12.44 - Equivalent model to Figure 12.43**

*Package CompleteActivities*

The figure below illustrates three examples of using the weight attribute. The Cricket example uses a constant weight to indicate that a cricket team cannot be formed until eleven players are present. The Task example uses a non-constant weight to indicate that an invoice for a particular job can only be sent when all of its tasks have been completed. The proposal example depicts an activity for placing bids for a proposal, where many such bids can be placed. Then, when the bidding period is over, the Award Proposal Bid activity reads all the bids as a single set and determines which vendor to award the bid.

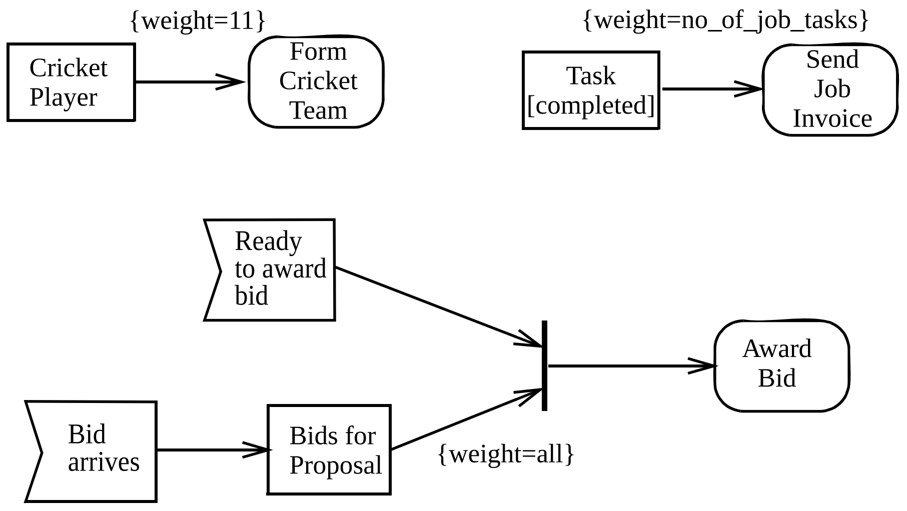


**Figure 12.45 - Activity edge examples**

## Rationale

Activity edges are introduced to provide a general class for connections between activity nodes.

### Changes from previous UML

ActivityEdge replaces the use of (state) Transition in UML 1.5 activity modeling. It also replaces data flow and control flow links in UML 1.5 action model.

## 12.3.6 ActivityFinalNode (from BasicActivities, IntermediateActivities)

An activity final node is a final node that stops all flows in an activity.

### Generalizations

- “ControlNode (from BasicActivities)” on page 346
- “FinalNode (from IntermediateActivities)” on page 360

### Description

An activity may have more than one activity final node. The first one reached stops all flows in the activity.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

A token reaching an activity final node terminates the activity (or structured node, see “StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)” on page 396). In particular, it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes. Terminating the execution of synchronous invocation actions also terminates whatever behaviors they are waiting on for return. Any behaviors invoked asynchronously by the activity are not affected. All tokens offered on the incoming edges are accepted. Any object nodes declared as outputs are passed out of the containing activity, using the null token for object nodes that have nothing in them. If there is more than one final node in an activity, the first one reached terminates the activity, including the flow going towards the other activity final.

If it is not desired to abort all flows in the activity, use flow final instead. 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 activity final. Using a flow final will simply consume the tokens reaching it without aborting other flows. Or arrange for separate invocations of the activity to use separate executions of the activity, so tokens from separate invocations will not affect each other.

## Notation

Activity final nodes are notated as a solid circle with a hollow circle, as indicated in the figure below. It can be thought of as a goal notated as “bull’s eye,” or target.

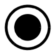


**Figure 12.46 - Activity final notation**

## Examples

The first example below depicts that when the Close Order behavior is completed, all tokens in the activity are terminated. This is indicated by passing control to an activity final node.

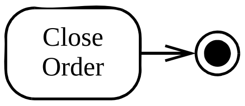


**Figure 12.47 - Activity final example**

The next figure is based on an example for an employee expense reimbursement process. It uses an activity diagram that illustrates two parallel flows racing to complete. The first one to reach the activity final aborts the others. The two flows appear in the same activity so they can share data. For example, who to notify in the case of no action.

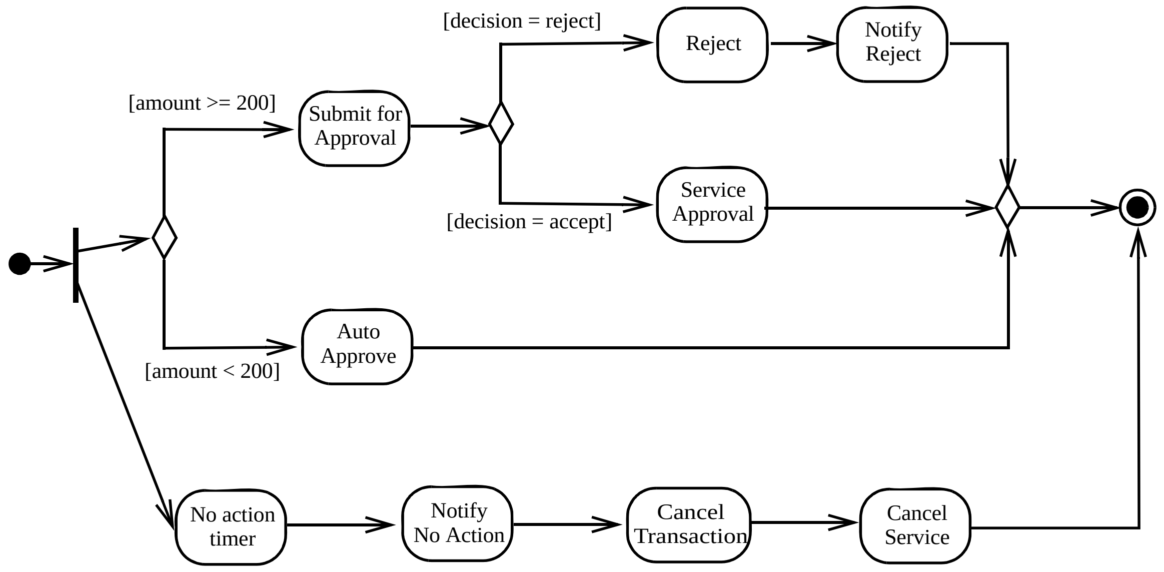


**Figure 12.48 - Activity final example**

In Figure 12.48, two ways to reach an activity final exist; but it is the result of exclusive “or” branching, not a “race” situation like the previous example. This example uses two activity final nodes, which has the same semantics as using one with two edges targeting it. The Notify of Modification behavior must not take long or the activity finals might kill it.

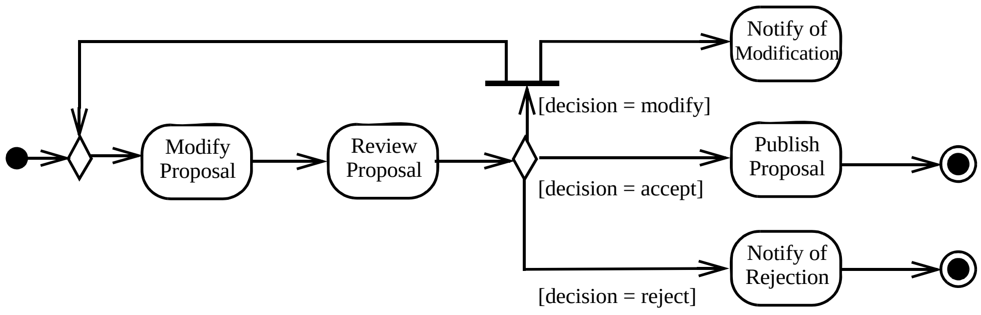


**Figure 12.49 - Activity final example**

### Rationale

Activity final nodes are introduced to model non-local termination of all flows in an activity.

### Changes from previous UML

ActivityFinal is new in UML 2.0.

## 12.3.7 ActivityGroup (from BasicActivities, FundamentalActivities)

(IntermediateActivities) An activity group is an abstract class for defining sets of nodes and edges in an activity.

### Generalizations

- “Element (from Kernel)” on page 60

### Description

Activity groups are a generic grouping construct for nodes and edges. Nodes and edges can belong to more than one group. They have no inherent semantics and can be used for various purposes. Subclasses of ActivityGroup may add semantics.

### Attributes

No additional attributes

### Associations

*Package FundamentalActivities*

| | |
|---|---|
| • activity : Activity [0..1] | Activity containing the group. |
| • containedNode : ActivityNode [0..*] | Nodes immediately contained in the group. |
| • /superGroup : ActivityGroup [0..1] | Group immediately containing the group. |
| • /subgroup : ActivityGroup [0..*] | Groups immediately contained in the group. |

*Package BasicActivities*

- containedEdge : ActivityEdge [0..*]  Edges immediately contained in the group.

### Constraints

[1] All nodes and edges of the group must be in the same activity as the group.

[2] No node or edge in a group may be contained by its subgroups or its containing groups, transitively.

[3] Groups may only be owned by activities or groups.

### Semantics

None

### Notation

No specific notation

### Rationale

Activity groups provide a generic grouping mechanism that can be used for various purposes, as defined in the subclasses of ActivityGroup, and in extensions and profiles.

### Changes from previous UML

ActivityGroups are new in UML 2.0.

## 12.3.8 ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, StructuredActivities)

An activity node is an abstract class for points in the flow of an activity connected by edges.

### Generalizations

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

### Description

An activity node is an abstract class for the steps of an activity. It covers executable nodes, control nodes, and object nodes.

(BasicActivities) Nodes can be replaced in generalization and (CompleteActivities) be contained in interruptible regions.

### Attributes

No additional attributes

### Associations

*Package FundamentalActivities*

- activity : Activity[0..1]  Activity containing the node.

- /inGroup : Group [0..*] Groups containing the node. Multiplicity specialized to [0..1] for StructuredActivityGroup.

*Package BasicActivities*

- incoming : ActivityEdge [0..*] Edges that have the node as target.
- outgoing : ActivityEdge [0..*] Edges that have the node as source.
- redefinedElement : ActivityNode [0..*] Inherited nodes replaced by this node in a specialization of the activity.

*Package IntermediateActivities*

- inPartition : Partition [0..*] Partitions containing the node.

*Package StructuredActivities*

- inStructuredNode : StructuredActivityNode [0..1] Structured activity node containing the node.

*Package CompleteActivities*

- inInterruptibleRegion : InterruptibleActivityRegion [0..*] Interruptible regions containing the node.

### Constraints

[1] Activity nodes can only be owned by activities or groups.

*Package StructuredActivities*

[1] Activity nodes may be owned by at most one structured node.

### Semantics

Nodes can be named, however, nodes are not required to have unique names within an activity to support multiple invocations of the same behavior or multiple uses of the same action. See Action, which is a kind of node. The fact that Activity is a Namespace, inherited through Behavior, does not affect this, because the containment of nodes is through ownedElement, the general ownership metaassociation for Element that does not imply unique names, rather than ownedMember. Other than naming, and functionality added by the complete version of activities, an activity node is only a point in an activity at this level of abstraction. See the children of ActivityNode for additional semantics.

*Package BasicActivities*

Nodes inherited from more general activities can be replaced. See RedefinableElement for more information on overriding inherited elements, and Activity for more information on activity generalization. See children of ActivityNode for additional semantics.

## Notation

The notations for activity nodes are illustrated below. There are three kinds of nodes: action node, object node, and control node. See these classes for more information.

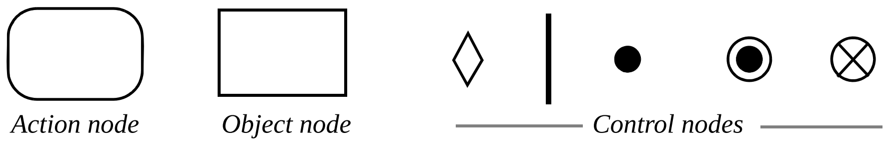


**Figure 12.50 - Activity node notation**

## Examples

This figure illustrates the following kinds of activity node: action nodes (e.g., Receive Order, Fill Order), object nodes (Invoice), and control nodes (the initial node before Receive Order, the decision node after Receive Order, and the fork node and Join node around Ship Order, merge node before Close Order, and activity final after Close Order).

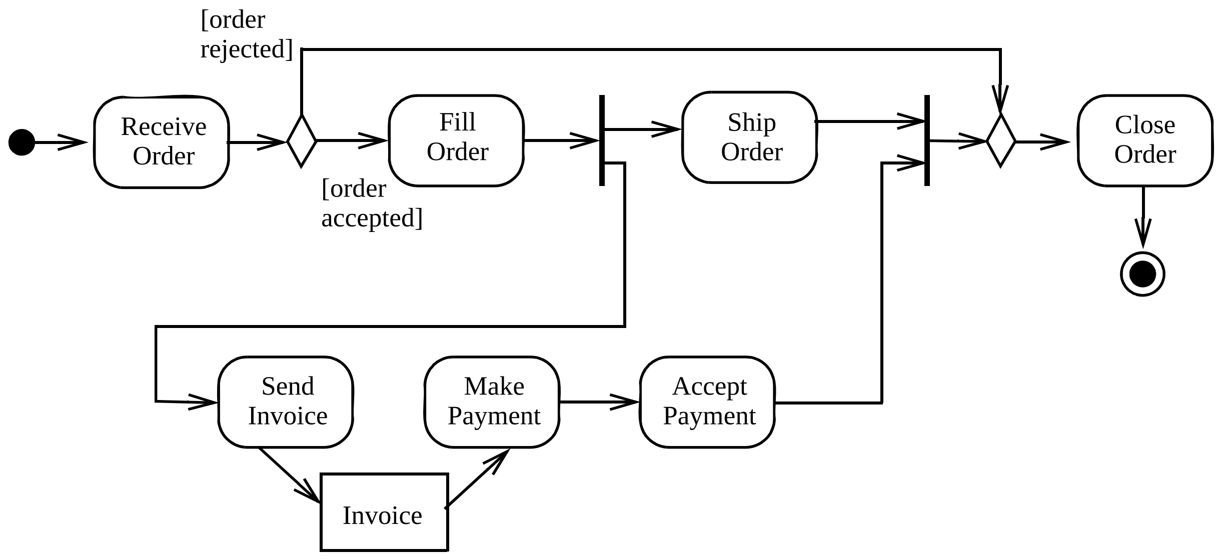


**Figure 12.51 - Activity node example (where the arrowed lines are only the non-activity node symbols)**

## Rationale

Activity nodes are introduced to provide a general class for nodes connected by activity edges.

## Changes from previous UML

ActivityNode replaces the use of StateVertex and its children for activity modeling in UML 1.5.

### 12.3.9 ActivityParameterNode (from BasicActivities)

An activity parameter node is an object node for inputs and outputs to activities.

#### Generalizations

- “ObjectNode (from BasicActivities, CompleteActivities)” on page 380

#### Description

Activity parameters are object nodes at the beginning and end of flows, to accept inputs to an activity and provide outputs from it.

Activity parameters inherit support for streaming and exceptions from Parameter.

#### Attributes

No additional attributes

#### Associations

- parameter : Parameter — The parameter the object node will be accepting and providing values for.

#### Constraints

[1] Activity parameter nodes must have parameters from the containing activity.

[2] The type of an activity parameter node is the same as the type of its parameter.

[3] Activity parameter nodes must have either no incoming edges or no outgoing edges.

[4] Activity parameter object nodes with no incoming edges and one or more outgoing edges must have a parameter with in or inout direction.

[5] Activity parameter object nodes with no outgoing edges and one or more incoming edges must have a parameter with out, inout, or return direction.

See “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” on page 301.

#### Semantics

When an activity is invoked, the inputs values are placed as tokens on the input activity parameter nodes, those with no incoming edges. Outputs of the activity must flow to output activity parameter nodes, those with no outgoing edges. See semantics at ObjectNode, Action, and ActivityParameterNode.

#### Notation

The label for parameter nodes can be a full specification of the corresponding parameter.

Also see notation at Activity.

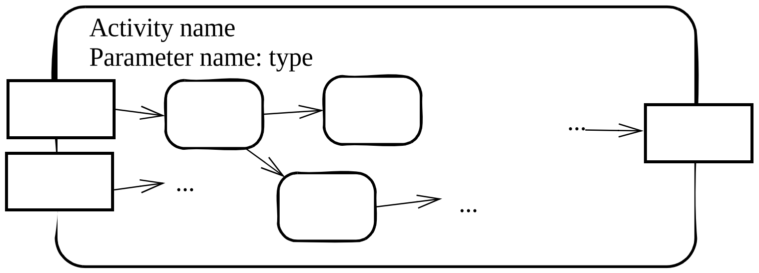


**Figure 12.52 - Activity notation**

The figure below shows annotations for streaming and exception activity parameters, which are the same as for pins. See Parameter for semantics of stream and exception parameters.

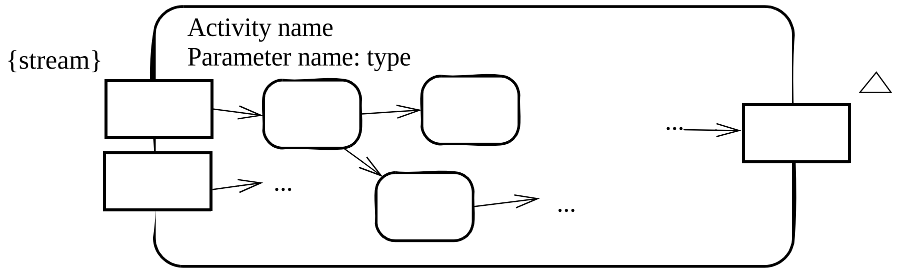


**Figure 12.53 - Activity notation**

## Presentation Options

If the round-cornered border of Figure 12.53 is replaced with the frame notation that is described in Annex A, then activity parameter nodes overlap the frame instead. If the round-cornered border or frame is omitted completely, then the activity parameter nodes can be placed anywhere, but it is clearer if they are placed in the same locations they would be in if the frame or border was shown.

The presentation option at the top of the activity diagram below may be used as notation for a model corresponding to the notation at the bottom of the diagram.

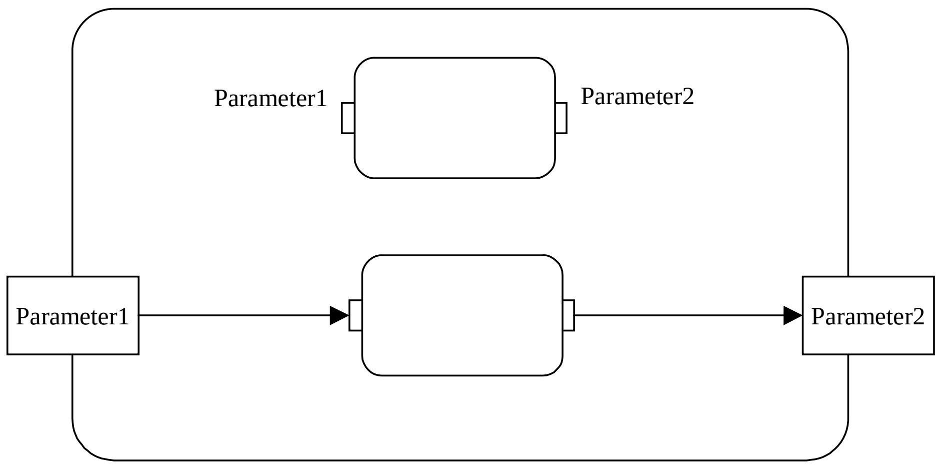


**Figure 12.54 - Presentation option for flows between pins and parameter nodes**

See presentation option for Pin when parameter is streaming. This can be used for activity parameters also.

### Examples

In the example below, production materials are fed into printed circuit board. At the end of the activity, computers are quality checked.

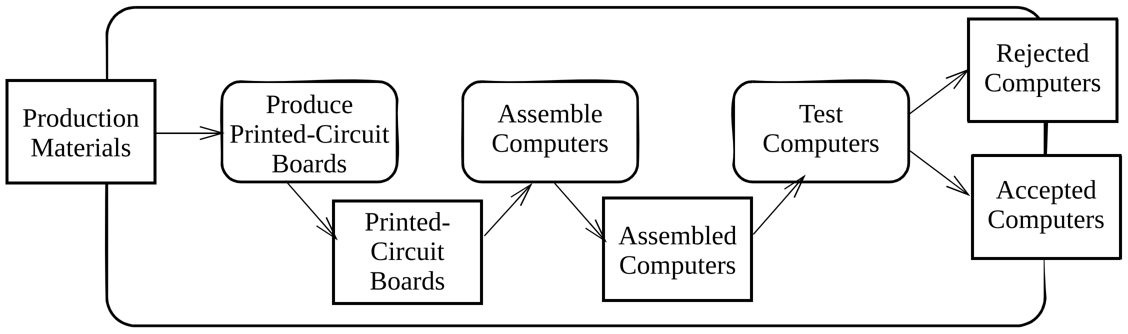


**Figure 12.55 - Example of activity parameters.nodes**

In the example below, production materials are streaming in to feed the ongoing printed circuit board fabrication. At the end of the activity, computers are quality checked. Computers that do not pass the test are exceptions. See Parameter for semantics of streaming and exception parameters.

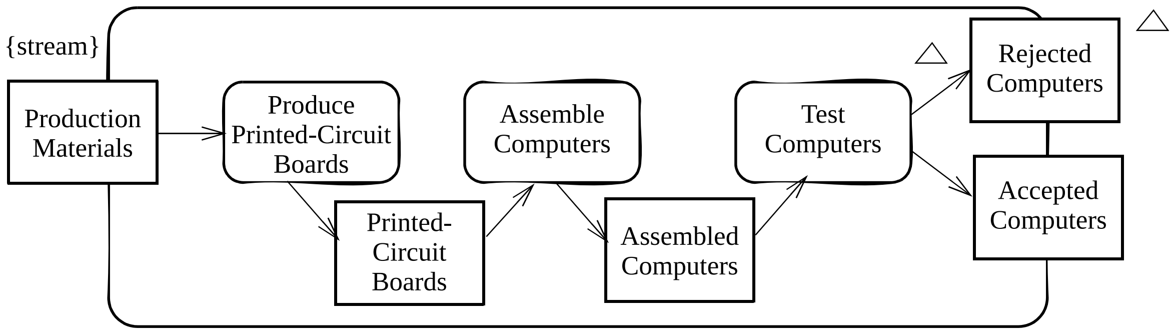


**Figure 12.56 - Example of activity parameter nodes for streaming and exceptions**

### Rationale

Activity parameter nodes are introduced to model parameters of activities in a way that integrates easily with the rest of the flow model.

### Changes from previous UML

ActivityParameterNode is new in UML 2.0.

## 12.3.10 ActivityPartition (from IntermediateActivities)

An activity partition is a kind of activity group for identifying actions that have some characteristic in common.

### Generalizations

- “ActivityGroup (from BasicActivities, FundamentalActivities)” on page 322
- “NamedElement (from Kernel, Dependencies)” on page 93

### Description

Partitions divide the nodes and edges to constrain and show a view of the contained nodes. Partitions can share contents. They often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the nodes of an activity.

### Attributes

| | |
|---|---|
| • isDimension : Boolean [1..1] = false | Tells whether the partition groups other partitions along a dimension. |
| • isExternal : Boolean [1..1] = false | Tells whether the partition represents an entity to which the partitioning structure does not apply. |

## Associations

| | |
|---|---|
| • superPartition : ActivityPartition [0..1] | Partition immediately containing the partition. Specializes *ActivityGroup::superGroup*. |
| • represents : Element [0..1] | An element constraining behaviors invoked by nodes in the partition. |
| • subgroup : ActivityPartition [0..*] | Partitions immediately contained in the partition. Specialized from *ActivityGroup::subgroup*. |
| • activity : Activity [0..1] | The activity containing the partition. Specialized from ActivityGroup. |

## Constraints

[1] A partition with isDimension = true may not be contained by another partition.

[2] No node or edge of a partition may be in another partition in the same dimension.

[3] If a partition represents a part, then all the non-external partitions in the same dimension and at the same level of nesting in that dimension must represent parts directly contained in the internal structure of the same classifier.

[4] If a non-external partition represents a classifier and is contained in another partition, then the containing partition must represent a classifier, and the classifier of the subpartition must be nested in the classifier represented by the containing partition, or be at the contained end of a strong composition association with the classifier represented by the containing partition.

[5] If a partition represents a part and is contained by another partition, then the part must be of a classifier represented by the containing partition, or of a classifier that is the type of a part representing the containing partition.

## Semantics

Partitions do not affect the token flow of the model. They constrain and provide a view on the behaviors invoked in activities. Constraints vary according to the type of element that the partition represents. The following constraints are normative:

***1) Classifier***

Behaviors of invocations contained by the partition are the responsibility of instances of the classifier represented by the partition. This means the context of invoked behaviors is the classifier. Invoked procedures containing a call to an operation or sending a signal must target objects at runtime that are instances of the classifier.

***2) Instance***

This imposes the same constraints as classifier, but restricted to a particular instance of the classifier.

***3) Part***

Behaviors of invocations contained by the partition are the responsibility of instances playing the part represented by the partition. This imposes the constraints for classifiers above according to the type of the part. In addition, invoked procedures containing a call to an operation or sending a signal must target objects at runtime that play the part at the time the message is sent. Just as partitions in the same dimension and nesting must be represented by parts of the same classifier's internal structure, all the runtime target objects of operation and signal passing invoked by the same execution of the activity must play parts of the same instance of the structured classifier. In particular, if an activity is executed in the context of a particular object at runtime, the parts of that object will be used as targets. If a part has more than one object playing it at runtime, the invocations are treated as if they were multiple, that is, the calls are sent in parallel, and the invocation does not complete until all the operations return.

***4) Attribute and Value***

A partition may be represented by an attribute and its subpartitions by values of that attribute. Behaviors of invocations contained by the subpartition have this attribute and the value represented by the subpartition. For example, a partition may represent the location at which a behavior is carried out, and the subpartitions would represent specific values for that attribute, such as Chicago. The location attribute could be on the process class associated with an activity, or added in a profile to extend behaviors with these attributes.

A partition may be marked as being a dimension for its subpartitions. For example, an activity may have one dimension of partitions for location at which the contained behaviors are carried out, and another for the cost of performing them. Dimension partitions cannot be contained in any other partition.

Elements other than actions that have behaviors or value specifications, such as transformation behaviors on edges, adhere to the same partition rules above for actions.

Partitions may be used in a way that provides enough information for review by high-level modelers, though not enough for execution. For example, if a partition represents a classifier, then behaviors in that partition are the responsibility of instances of the classifier, but the model may or may not say which instance in particular. In particular, a behavior in the partition calling an operation would be limited to an operation on that classifier, but an input object flow to the invocation might not be specified to tell which instance should be the target at runtime. The object flow could be specified in a later stage of development to support execution. Another option would be to use partitions that represent parts. Then when the activity executes in the context of a particular object, the parts of that object at runtime will be used as targets for the operation calls, as described above.

External partitions are intentional exceptions to the rules for partition structure. For example, a dimension may have partitions showing parts of a structured classifier. It can have an external partition that does not represent one of the parts, but a completely separate classifier. In business modeling, external partitions can be used to model entities outside a business.

## Notation

Activity partition may be indicated with two, usually parallel lines, either horizontal or vertical, and a name labeling the partition in a box at one end. Any activity nodes and edges placed between these lines are considered to be contained within the partition. Swimlanes can express hierarchical partitioning by representing the children in the hierarchy as further partitioning of the parent partition, as illustrated in b), below. Diagrams can also be partitioned multidimensionally, as depicted in c), below, where, each swim cell is an intersection of multiple partitions. The specification for each dimension (e.g., part, attribute) is expressed in next to the appropriate partition set.

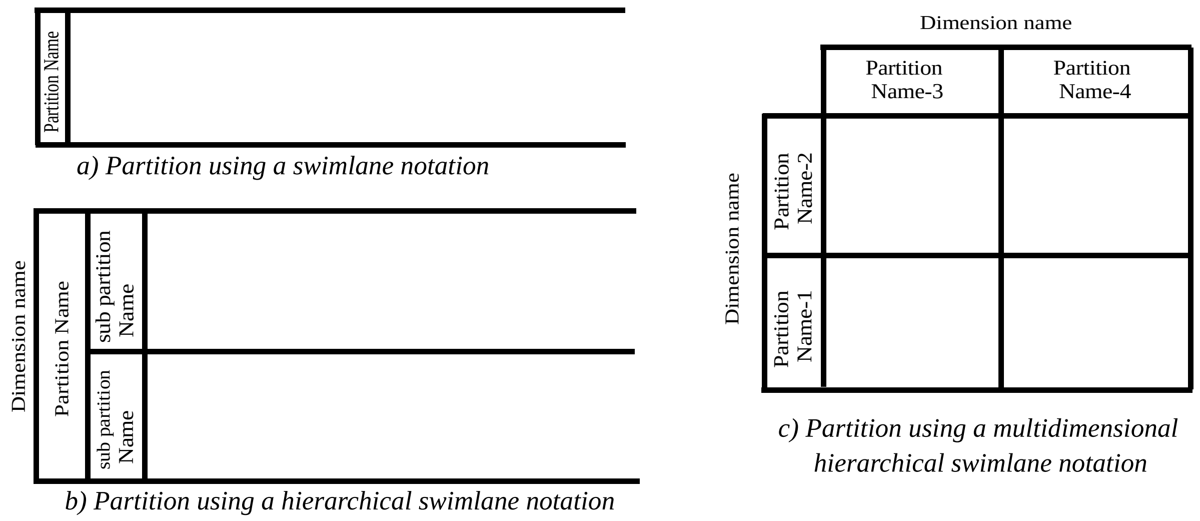


**Figure 12.57 - Activity partition notations**

In some diagramming situations, using parallel lines to delineate partitions is not practical. An alternate is to place the partition name in parenthesis above the activity name, as illustrated for actions in a), below. A comma-delimited list of partition names means that the node is contained in more than one partition. A double colon within a partition name indicates that the partition is nested, with the larger partitions coming earlier in the name. When activities are considered to occur outside the domain of a particular model, the partition can be labeled with the keyword «external», as illustrated in b) below. Whenever an activity in a swimlane is marked «external», this overrides the swimlane and dimension designation.

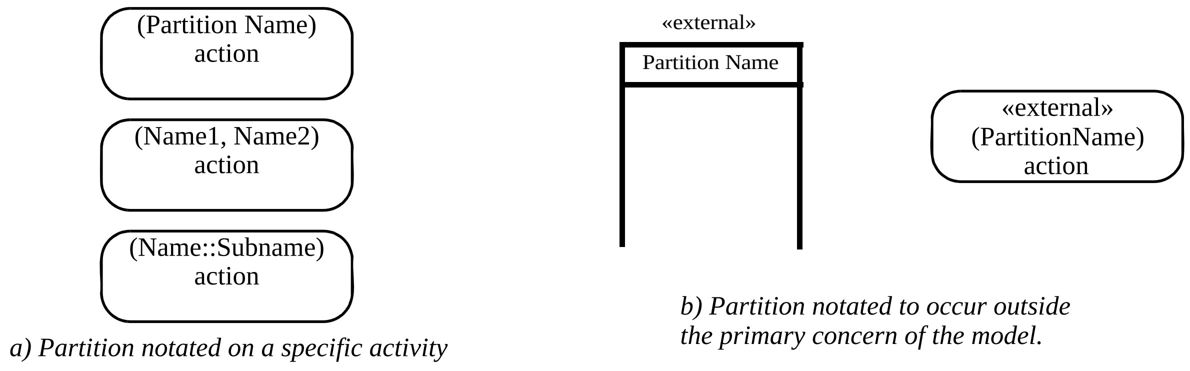


**Figure 12.58 - Activity partition notations**

## Presentation Options

When partitions are combined with the frame notation for Activity, the outside edges of the top level partition can be merged with the activity frame.

## Examples

The figures below illustrate an example of partitioning the order processing activity diagram into “swim lanes.” The top partition contains the portion of an activity for which the Order Department is responsible; the middle partition, the Accounting Department, and the bottom the Customer. These are attributes of the behavior invoked in the partitions, except for Customer, which is external to the domain. The flow of the invoice is not a behavior, so it does not need to appear in a partition.

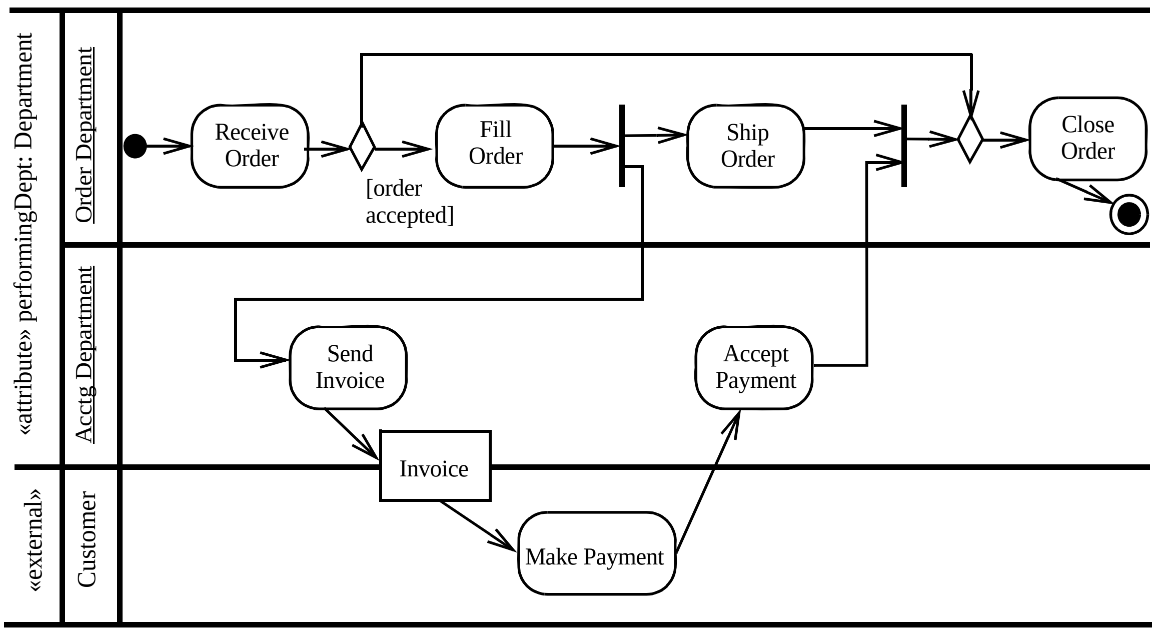


**Figure 12.59 - Activity partition using swimlane example**

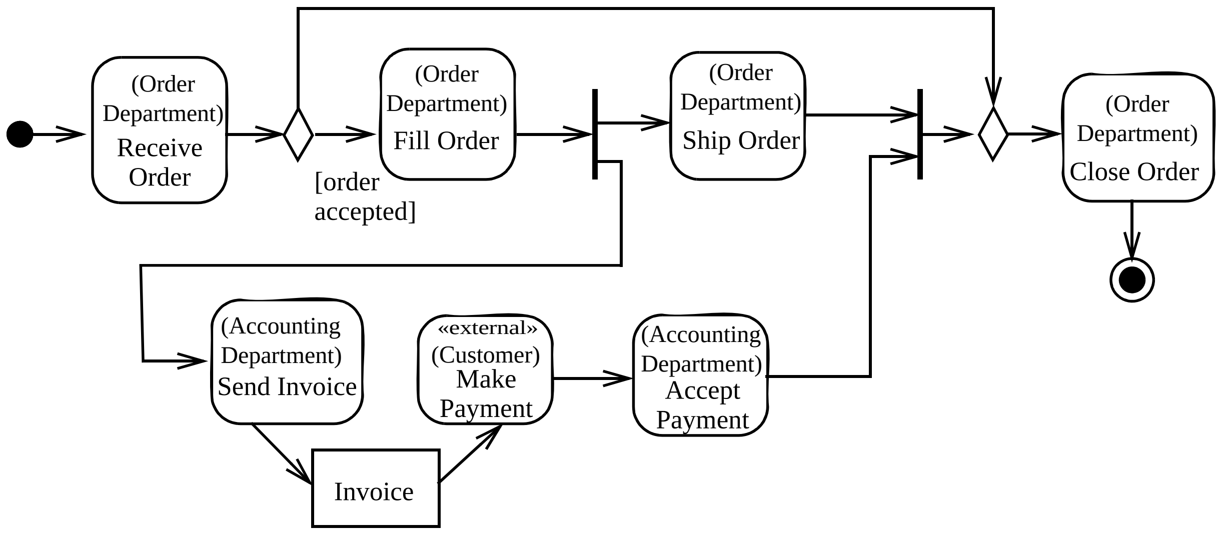

(Order Department) Receive Order
[order accepted]
(Order Department) Fill Order
(Order Department) Ship Order
(Order Department) Close Order
(Accounting Department) Send Invoice
«external» (Customer) Make Payment
(Accounting Department) Accept Payment
Invoice

**Figure 12.60 - Activity partition using annotation example**

The example below depicts multidimensional swim lanes. The Receive Order and Fill Order behaviors are performed by an instance of the Order Processor class, situated in Seattle, but not necessarily the same instance for both behaviors. Even though the Make Payment is contained within the Seattle/Accounting Clerk swim cell, its performer and location are not specified by the containing partition, because it has an overriding partition.

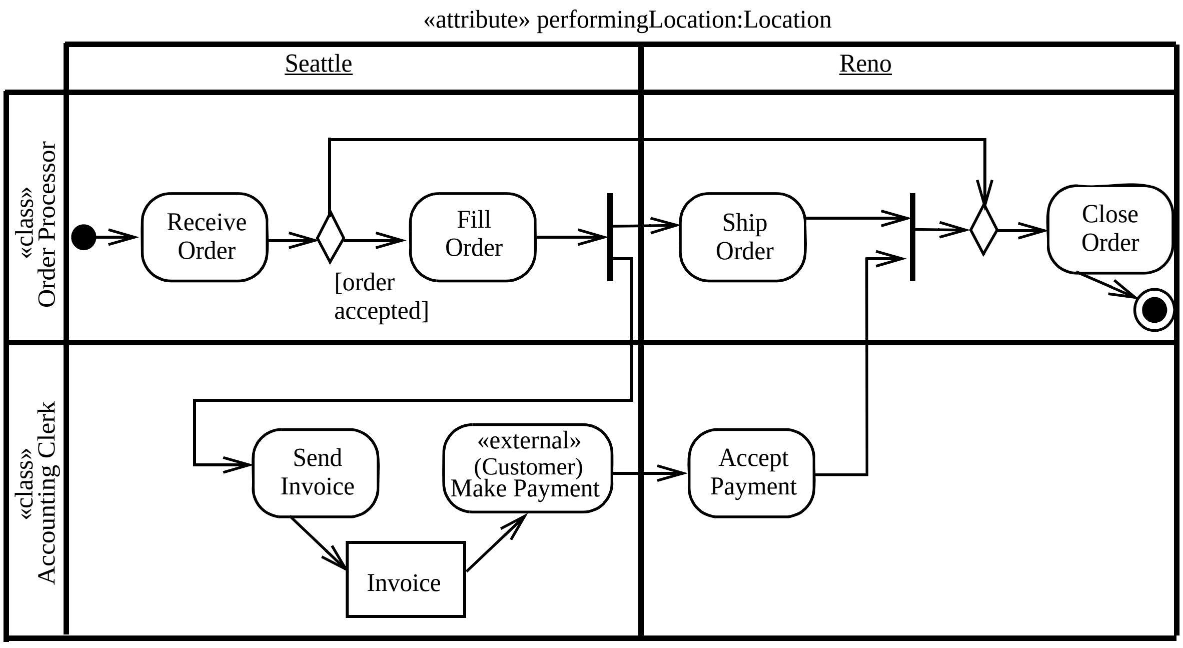


**Figure 12.61 - Activity partition using multidimensional swimlane example**

### Rationale

Activity partitions are introduced to support the assignment of domain-specific information to nodes and edges.

### Changes from previous UML

Edges can be contained in partitions in UML 2.0. Additional notation is provided for cases when swimlanes are too cumbersome. Partitions can be hierarchical and multidimensional. The relation to classifier, parts, and attributes is formalized, including external partitions as exceptions to these rules.

## 12.3.11 AddVariableValueAction (as specialized)

See "AddVariableValueAction (from StructuredActions)" on page 234.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

See "AddVariableValueAction (from StructuredActions)" on page 234.

### Notation

### Presentation Options

The presentation option at the top of Figure 12.62 may be used as notation for a model corresponding to the notation at the bottom of the figure. If the action has non-defaulted metaattribute values, these can be shown with a property list near the variable name.

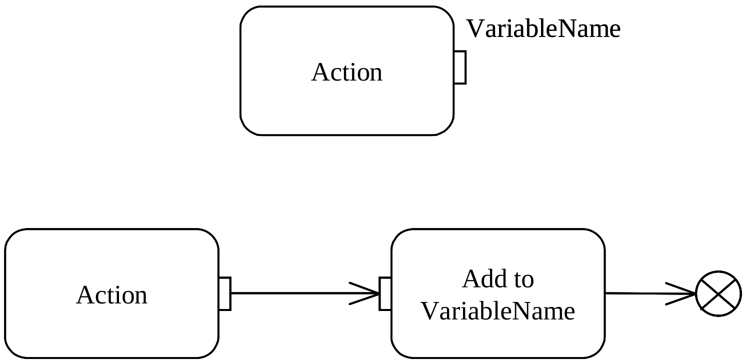


**Figure 12.62 - Presentation option for AddVariableValueAction**

### 12.3.12 Behavior (from CompleteActivities)

Behavior is specialized to own zero or more ParameterSets.

#### Generalizations

- “Behavior (from BasicBehaviors)” on page 416 *(merge increment)*.

#### Description

The concept of Behavior is extended to own ParameterSets.

#### Attributes

No additional attributes.

#### Associations

- ownedParameterSets : ParameterSet[0..*] The ParameterSets owned by this Behavior.

#### Constraints

See “ParameterSet (from CompleteActivities)” on page 386.

#### Semantics

See semantics of “ParameterSet (from CompleteActivities)” on page 386.

#### Notation

See notation for “ParameterSet (from CompleteActivities)” on page 386.

#### Examples

See examples for “ParameterSet (from CompleteActivities)” on page 386.

#### Changes from previous UML

ParameterSet is new in UML 2.0.

### 12.3.13 BehavioralFeature (from CompleteActivities)

BehavioralFeature is specialized to own zero or more ParameterSets.

#### Generalizations

- “BehavioralFeature (from BasicBehaviors, Communications)” on page 418 *(merge increment)*.

#### Description

The concept of BehavioralFeature is extended to own ParameterSets.

### Attributes

No additional attributes

### Associations

- ownedParameterSets : ParameterSet[0..*] The ParameterSets owned by this BehavioralFeature.

### Constraints

See “ParameterSet (from CompleteActivities)” on page 386.

### Semantics

See semantics of “ParameterSet (from CompleteActivities)” on page 386.

### Notation

See notation for “ParameterSet (from CompleteActivities)” on page 386.

### Examples

See examples for “ParameterSet (from CompleteActivities)” on page 386.

### Changes from previous UML

ParameterSet is new in UML 2.0.

## 12.3.14 CallBehaviorAction (as specialized)

“CallBehaviorAction (from BasicActions)” on page 237

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

[1] When all the control and data flow prerequisites of the action execution are satisfied, CallBehaviorAction consumes its input tokens and invokes its specified behavior. The values in the input tokens are made available to the invoked behavior as argument values. When the behavior is finished, tokens are offered on all outgoing control edges, with a copy made for each control edge. Object and data tokens are offered on the outgoing object flow edges as determined by the output pins. Each parameter of the behavior of the action provides output to a pin or takes input from one. See Pin. The inputs to the action determine the actual arguments of the call.

[2] If the call is asynchronous, a control token is offered to each outgoing control edge of the action and execution of the action is complete. Execution of the invoked behavior proceeds without any further dependency on the execution of the

activity containing the invoking action. Once the invocation of the behavior has been initiated, execution of the asynchronous action is complete.

[3] An asynchronous invocation completes when its behavior is started, or is at least ensured to be started at some point. When an asynchronous invocation is done, the flow continues regardless of the status of the invoked behavior. Any return or out values from the invoked behavior are not passed back to the containing activity. For example, the containing activity may complete even though the invoked behavior is not finished. This is why asynchronous invocation is not the same as using a fork to invoke the behavior followed by a flow final. A forked behavior still needs to finish for the containing activity to finish. If it is desired to complete the invocation, but have some outputs provided later when they are needed, then use a fork to give the invocation its own flow line, and rejoin the outputs of the invocation to the original flow when they are needed.

[4] If the call is synchronous, execution of the calling action is blocked until it receives a reply token from the invoked behavior. The reply token includes values for any return, out, or inout parameters.

[5] If the call is synchronous, when the execution of the invoked behavior completes, the result values are placed as object tokens on the result pins of the call behavior action, a control token is offered on each outgoing control edge of the call behavior action, and the execution of the action is complete. (StructuredActions, ExtraStructuredActivities) If the execution of the invoked behavior yields an exception, the exception is transmitted to the call behavior action to begin the search for the handler. See “RaiseExceptionAction (from StructuredActions)” on page 258.

### Notation

The name of the behavior, or other description of it, that is performed by the action is placed inside the rectangle. If the node name is different than the behavior name, then it appears in the symbol instead. Pre- and postconditions on the behavior can be shown similarly to Figure 12.29 on page 303, using keywords «precondition» and «postcondition».

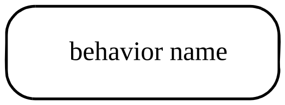


**Figure 12.63 CallBehaviorAction**

The call of an activity is indicated by placing a rake-style symbol within the symbol. The rake resembles a miniature hierarchy, indicating that this invocation starts another activity that represents a further decomposition. An alternative notation in the case of an invoked activity is to show the contents of the invoked activity inside a large round-cornered rectangle. Edges flowing into the invocation connect to the parameter object nodes in the invoked activity. The parameter object nodes are shown on the border of the invoked activity. The model is the same regardless of the choice of notation. This assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

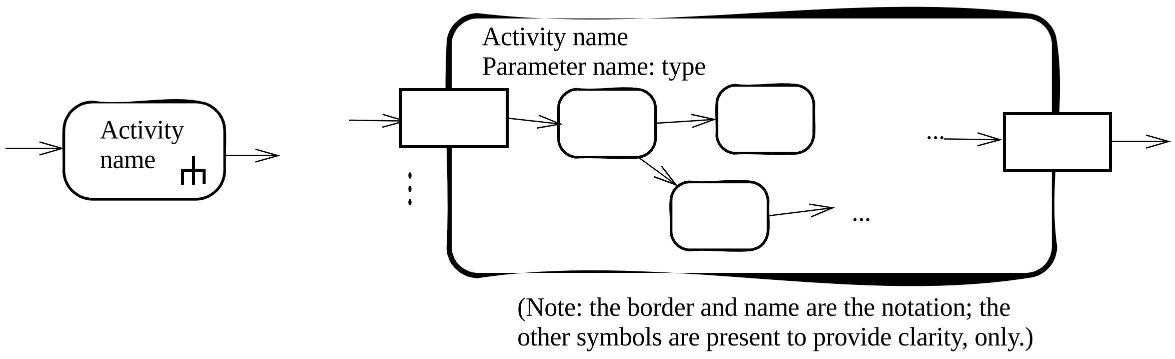

Activity
name
Activity name
Parameter name: type
...
...
(Note: the border and name are the notation; the
other symbols are present to provide clarity, only.)

**Figure 12.64 - Invoking Activities that have nodes and edges**

Below is an example of invoking an activity called FillOrder.

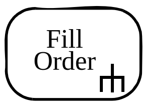


**Figure 12.65 - Example of invoking an activity**

### Rationale

"CallBehaviorAction (from BasicActions)" on page 237

### Changes from previous UML

"CallBehaviorAction (from BasicActions)" on page 237

## 12.3.15 CallOperationAction (as specialized)

See "CallOperationAction (from BasicActions)" on page 239.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

See "CallOperationAction (from BasicActions)" on page 239.

### Notation

The name of the operation, or other description of it, is displayed in the symbol. Pre- and postconditions on the operation can be shown similarly to Figure 12.29 on page 303, using keywords «precondition» and «postcondition».

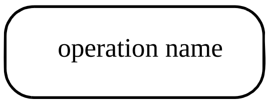


**Figure 12.66 - Calling an operation**

### Presentation Options

If the node has a different name than the operation, then this is used in the symbol instead. The name of the class may optionally appear below the name of the operation, in parentheses postfixed by a double colon. If the node name is different than the operation name, then the behavioral feature name may be shown after the double colon.

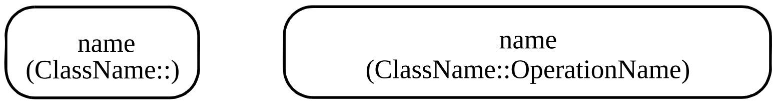


**Figure 12.67 - Invoking behavioral feature notations**

### Rationale

See “CallOperationAction (from BasicActions)” on page 239.

### Changes from previous UML

See “CallOperationAction (from BasicActions)” on page 239.

## 12.3.16 CentralBufferNode (from IntermediateActivities)

A central buffer node is an object node for managing flows from multiple sources and destinations.

### Generalizations

- “ObjectNode (from BasicActivities, CompleteActivities)” on page 380

### Description

A central buffer node accepts tokens from upstream object nodes and passes them along to downstream object nodes. They act as a buffer for multiple in flows and out flows from other object nodes. They do not connect directly to actions.

### Attributes

No additional attributes

## Associations

No additional associations

## Semantics

See semantics at ObjectNode. All object nodes have buffer functionality, but central buffers differ in that they are not tied to an action as pins are, or to an activity as activity parameter nodes are. See example below.

## Notation

See notation at ObjectNode. A central buffer may also have the keyword «centralBuffer» as shown below. This is useful when it needs to be distinguished from the standalone notation for pins shown on the left of Figure 12.119 and the top left of Figure 12.126.

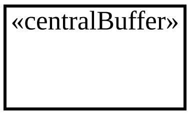


**Figure 12.68 - Optional central buffer notation**

## Examples

In the example below, the behaviors for making parts at two factories produce finished parts. The central buffer node collects the parts, and behaviors after it in the flow use them as needed. All the parts that are not used will be packed as spares, and vice versa, because each token can only be drawn from the object node by one outgoing edge. The choice in this example is non-deterministic.

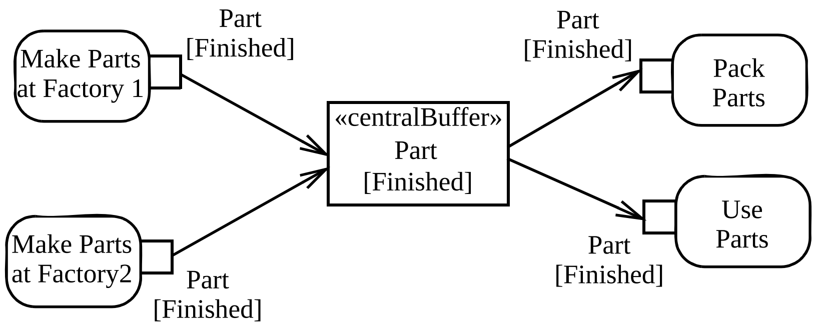


**Figure 12.69 - Central buffer node example**

## Rationale

Central buffer nodes give additional support for queuing and competition between flowing objects.

## Changes from previous UML

CentralBufferNode is new in UML 2.0.

## 12.3.17 Clause (from CompleteStructuredActivities, StructuredActivities)

### Generalizations

- "Element (from Kernel)" on page 60

### Description

A clause is an element that represents a single branch of a conditional construct, including a test and a body section. The body section is executed only if (but not necessarily if) the test section evaluates true.

### Attributes

No additional attributes

### Associations

*Package StructuredActivities*

| | |
|---|---|
| • test : ActivityNode [0..*] | A nested activity fragment with a designated output pin that specifies the result of the test. |
| • body : ActivityNode [0..*] | A nested activity fragment that is executed if the test evaluates to true and the clause is chosen over any concurrent clauses that also evaluate to true. |
| • predecessorClause : Clause [*] | A set of clauses whose tests must all evaluate false before the current clause can be tested. |
| • successorClause : Clause [*] | A set of clauses that may not be tested unless the current clause tests false. |
| • decider : OutputPin [1] | An output pin within the test fragment the value of which is examined after execution of the test to determine whether the body should be executed. |

*Package CompleteStructuredActivities*

| | |
|---|---|
| • bodyOutput : OutputPin [0..*] | A list of output pins within the body fragment whose values are copied to the result pins of the containing conditional node or conditional node after execution of the clause body. |

### Semantics

The semantics are explained under "ConditionalNode (from CompleteStructuredActivities, StructuredActivities)."

## 12.3.18 ConditionalNode (from CompleteStructuredActivities, StructuredActivities)

A conditional node is a structured activity node that represents an exclusive choice among some number of alternatives.

### Generalizations

- "StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)" on page 396

### Description

A conditional node consists of one or more clauses. Each clause consists of a test section and a body section. When the conditional node begins execution, the test sections of the clauses are executed. If one or more test sections yield a true value, one of the corresponding body sections will be executed. If more than one test section yields a true value, only one body section will be executed. The choice is nondeterministic unless the test sequence of clauses is specified. If no test section yields a true value, then no body section is executed; this may be a semantic error if output values are expected from the conditional node.

In general, test section may be executed in any order, including simultaneously (if the underlying execution architecture supports it). The result may therefore be nondeterministic if more than one test section can be true concurrently. To enforce ordering of evaluation, sequencing constraints may be specified among clauses. One frequent case is a total ordering of clauses, in which case the result is determinate. If it is impossible for more than one test section to evaluate true simultaneously, the result is deterministic and it is unnecessary to order the clauses, as ordering may impose undesirable and unnecessary restrictions on implementation. Note that, although evaluation of test sections may be specified as concurrent, this does not require that the implementation evaluate them in parallel; it merely means that the model does not impose any order on evaluation.

An “else” clause is a clause that is a successor to all other clauses in the conditional and whose test part always returns true. A notational gloss is provided for this frequent situation.

Output values created in the test or body section of a clause are potentially available for use outside the conditional. However, any value used outside the conditional must be created in every clause, otherwise an undefined value would be accessed if a clause not defining the value were executed.

### Attributes

*Package StructuredActivities*

| | |
|---|---|
| • isAssured : Boolean | If true, the modeler asserts that at least one test will succeed. |
| • isDeterminate: Boolean | If true, the modeler asserts that at most one test will succeed concurrently and therefore the choice of clause is deterministic. |

### Associations

*Package StructuredActivities*

| | |
|---|---|
| • clause : Clause[1..*] | Set of clauses composing the conditional. |

*Package CompleteStructuredActivities*

| | |
|---|---|
| • result : OutputPin [0..*] | A list of output pins that constitute the data flow outputs of the conditional. |

### Constraints

No additional constraints

### Semantics

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

The test section of any clause without a predecessorClause is eligible for execution immediately. If a test section yields a false value, a control token is delivered to all of its successorClauses. Any test section with a predecessorClause is eligible for execution when it receives control tokens from each of its predecessor clauses.

If a test section yields a true value, then the corresponding body section is executed provided another test section does not also yield a true value. If more than one test section yields a true value, exactly one body section will be executed, but it is indeterminate which one will be executed. When a body section is chosen for execution, the evaluation of all other test parts is terminated (just like an interrupting edge). If some of the test parts have external effects, terminating them may be another source of indeterminacy. Although test parts are permitted to produce side effects, avoiding side effects in tests will greatly reduce the chance of logical errors and race conditions in a model and in any code generated from it.

If no test section yields a true value, the execution of the conditional node terminates with no outputs. This may be a semantic error if a subsequent node requires an output from the conditional. It is safe if none of the clauses create outputs. If the *isAssured* attribute of the conditional node has a true value, the modeler asserts that at least one test section will yield a test value. If the *isDeterminate* attribute has a true value, the modeler asserts that at most one test section will concurrently yield a test value (the predecessor relationship may be used to enforce this assertion). Note that it is, in general, impossible for a computer system to verify these assertions, so they may provide useful information to a code generator, but if the assertions are incorrect, then incorrect code may be generated.

When a body section is chosen for execution, all of its nodes without predecessor flows within the conditional receive control tokens and are enabled for execution. When execution of all nodes within the body section has completed, execution of the conditional node is complete and its successors are enabled.

Within the body section, variables defined in the loop node or in some higher-level enclosing node may be accessed and updated with new values. Values that are used in a data flow manner must be created or updated in all clauses of the conditional, otherwise undefined values would be accessed.

#### Notation

No specific notation.

#### Style Guidelines

Mixing sequential and concurrent tests in one conditional may be confusing, although it is permitted.

#### Rationale

Conditional nodes are introduced to provide a structured way to represent decisions.

#### Changes from previous UML

Conditional nodes replace ConditionalAction from the UML 1.5 action model.

### 12.3.19 ControlFlow (from BasicActivities)

A control flow is an edge that starts an activity node after the previous one is finished.

#### Generalizations

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

#### Description

Objects and data cannot pass along a control flow edge.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

[1] Control flows may not have object nodes at either end, except for object nodes with control type.

## Semantics

See semantics inherited from ActivityEdge. A control flow is an activity edge that only passes control tokens. Tokens offered by the source node are all offered to the target node.

## Notation

A control flow is notated by an arrowed line connecting two actions.

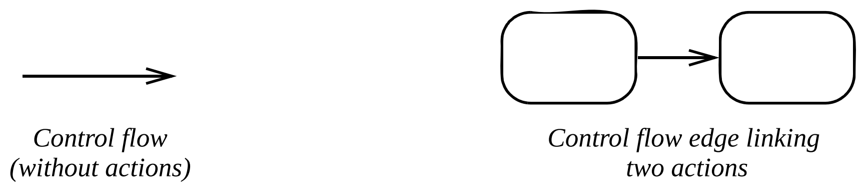


**Figure 12.70 - Control flow notation**

## Examples

The figure below depicts an example of the Fill Order action passing control to the Ship Order action. The activity edge between the two is a control flow, which indicates that when Fill Order is completed, Ship Order is invoked.

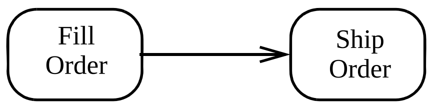


**Figure 12.71 - Control flow example**

## Rationale

Control flow is introduced to model the sequencing of behaviors that does not involve the flow of objects.

## Changes from previous UML

Explicitly modeled control flows are new to activity modeling in UML 2.0. They replace the use of (state) Transition in UML 1.5 activity modeling. They replace control flows in UML 1.5 action model.

## 12.3.20 ControlNode (from BasicActivities)

A control node is an abstract activity node that coordinates flows in an activity.

### Generalizations

- "ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, StructuredActivities)" on page 323.

### Description

A control node is an activity node used to coordinate the flows between other nodes. It covers initial node, final node and its children, fork node, join node, decision node, and merge node.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

See semantics at Activity. See subclasses for the semantics of each kind of control node.

### Notation

The notations for control nodes are illustrated below: decision node, initial node, activity final, and flow final.

Fork node and join node are the same symbol, they have different semantics and are distinguished notationally by the way edges are used with them. For more information, see ForkNode and JoinNode below.

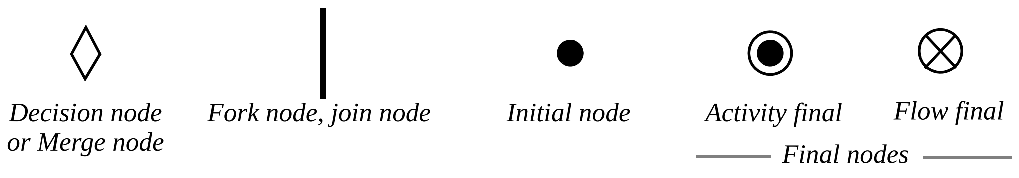


**Figure 12.72 - Control node notations**

### Examples

The figure below contains examples of various kinds of control nodes. An initial node is depicted in the upper left as triggering the Receive Order action. A decision node after Received Order illustrates branching based on order rejected or order accepted conditions. Fill Order is followed by a fork node that passes control both to Send Invoice and Ship Order.

The join node indicates that control will be passed to the merge when both Ship Order and Accept Payment are completed. Since a merge will just pass the token along, Close Order activity will be invoked. (Control is also passed to Close Order whenever an order is rejected.) When Close Order is completed, control passes to an activity final.

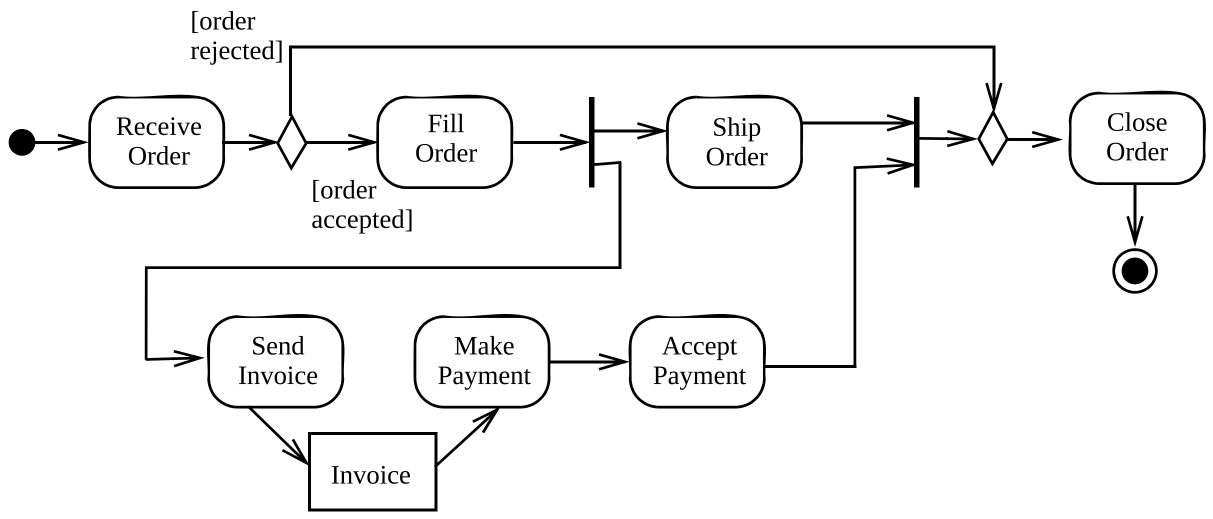


**Figure 12.73 - Control node examples (with accompanying actions and control flows)**

### Rationale

Control nodes are introduced to provide a general class for nodes that coordinate flows in an activity.

### Changes from previous UML

ControlNode replaces the use of PseudoState in UML 1.5 activity modeling.

## 12.3.21 DataStoreNode (from CompleteActivities)

A data store node is a central buffer node for non-transient information.

### Generalizations

- “CentralBufferNode (from IntermediateActivities)” on page 340

### Description

A data store keeps all tokens that enter it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any tokens in the object node containing that object.

### Attributes

No additional attributes

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

Tokens chosen to move downstream are copied so that tokens appear to never leave the data store. If a token containing an object is chosen to move into a data store, and there is a token containing that object already in the data store, then the chosen token replaces the existing one. Selection and transformation behavior on outgoing edges can be designed to get information out of the data store, as if a query were being performed. For example, the selection behavior can identify an object to retrieve and the transformation behavior can get the value of an attribute on that object. Selection can also be designed to only succeed when a downstream action has control passed to it, thereby implementing the pull semantics of earlier forms of data flow.

## Notation

The data store notation is a special case of the object node notation, using the label «datastore».

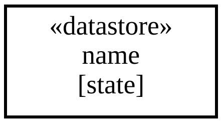


**Figure 12.74 - Data store node notation.**

## Examples

The figure below is an example of using a data store node.

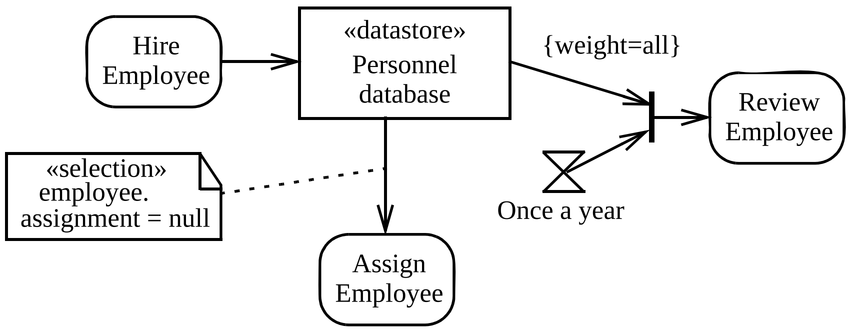

Hire Employee
«datastore» Personnel database
{weight=all}
Review Employee
«selection» employee. assignment = null
Once a year
Assign Employee

**Figure 12.75 - Data store node example**

## Rationale

Data stores are introduced to support earlier forms of data flow modeling in which data is persistent and used as needed, rather than transient and used when available.

### Changes from previous UML

Data stores are new in UML 2.0.

## 12.3.22 DecisionNode (from IntermediateActivities)

A decision node is a control node that chooses between outgoing flows.

### Generalizations

- “ControlNode (from BasicActivities)” on page 346

### Description

A decision node has one incoming edge and multiple outgoing activity edges.

### Attributes

No additional attributes

### Associations

| | |
|---|---|
| • decisionInput : Behavior [0..1] | Provides input to guard specifications on edges outgoing from the decision node. |

### Constraints

[1] A decision node has one incoming edge.

[2] A decision input behavior has zero or one input parameter and one output parameter. Any input parameter must be the same as or a supertype of the type of object tokens coming along the incoming edge. The behavior cannot have side effects.

[3] The edges coming into and out of a decision node must be either all object flows or all control flows.

### Semantics

Each token arriving at a decision node can traverse only one outgoing edge. Tokens are not duplicated. Each token offered by the incoming edge is offered to the outgoing edges.

Most commonly, guards of the outgoing edges are evaluated to determine which edge should be traversed. The order in which guards are evaluated is not defined, because edges in general are not required to determine which tokens they accept in any particular order. The modeler should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges. If the implementation can ensure that only one guard will succeed, it is not required to evaluate all guards when one is found that does. For decision points, a predefined guard “else” may be defined for at most one outgoing edge. This guard succeeds for a token only if the token is not accepted by all the other edges outgoing from the decision point.

Notice that the semantics only requires that the token traverse one edge, rather than be offered to only one edge. Multiple edges may be offered the token, but if only one of them has a target that accepts the token, then that edge is traversed. If multiple edges accept the token and have approval from their targets for traversal at the same time, then the semantics is not defined.

If a decision input behavior is specified, then each data token is passed to the behavior before guards are evaluated on the outgoing edges. The behavior is invoked without input for control tokens. The output of the behavior is available to the guard. Because the behavior is used during the process of 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. It may not modify objects, but it may for example, navigate from one object to another or get an attribute value from an object.

### Notation

The notation for a decision node is a diamond-shaped symbol, as illustrated on the left side of the figure below. Decision input behavior is specified by the keyword «decisionInput» placed in a note symbol, and attached to the appropriate decision node symbol as illustrated in the figure below.

A decision node must have a single activity edge entering it, and one or more edges leaving it. The functionality of decision node and merge 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 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 RFP supports the interchange of diagram elements and their mapping to model elements.

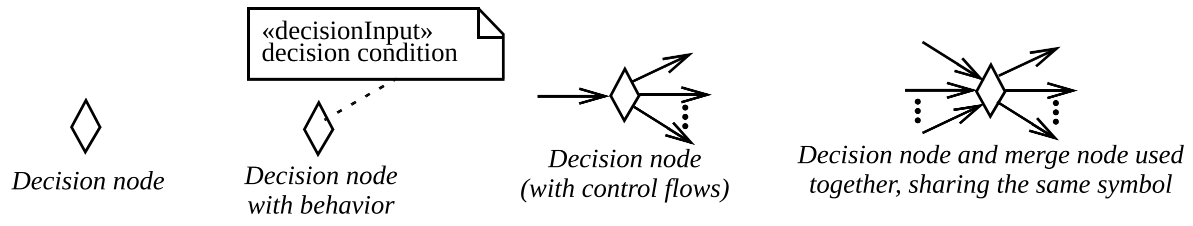

«decisionInput»
decision condition
Decision node
Decision node with behavior
Decision node (with control flows)
Decision node and merge node used together, sharing the same symbol

**Figure 12.76 - Decision node notation**

### Examples

The figure below contains a decision node that follows the Received Order behavior. The branching is based on whether order was rejected or accepted. An order accepted condition results in passing control to Fill Order and rejected orders to Close Order.

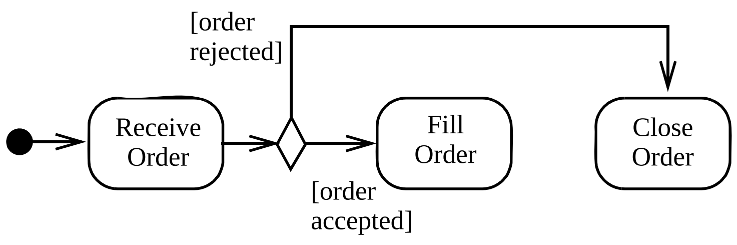


**Figure 12.77 - Decision node example**

The example in the figure below illustrates an order process example. Here, an order item is pulled from stock and prepared for delivery. Since the item has been removed from inventory, the reorder level should also be checked; and if the actual level falls below a pre-specified reorder point, more of the same type of item should be reordered.

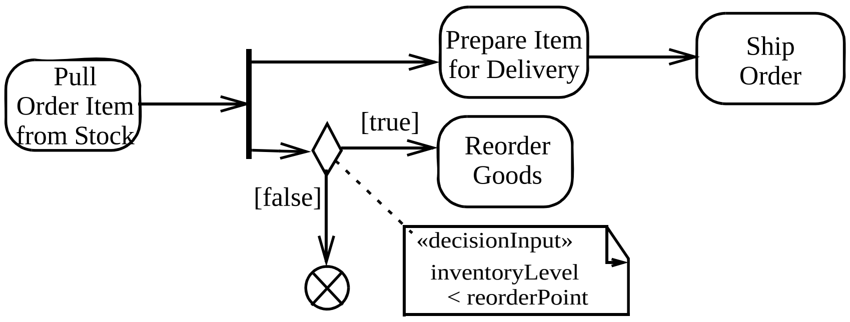

Prepare Item for Delivery
Ship Order
Pull Order Item from Stock
[true]
Reorder Goods
[false]
«decisionInput»
inventoryLevel < reorderPoint

**Figure 12.78 - Decision node example**

### Rationale

Decision nodes are introduced to support conditionals in activities. Decision input behaviors are introduced to avoid redundant recalculations in guards.

### Changes from previous UML

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

## 12.3.23 ExceptionHandler (from ExtraStructuredActivities)

### Generalizations

- “Element (from Kernel)” on page 60

### Description

An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node.

### Associations

| | |
|---|---|
| • protectedNode : ExecutableNode [1..1] | The node protected by the handler. The handler is examined if an exception propagates to the outside of the node. |
| • handlerBody : ExecutableNode [1..1] | A node that is executed if the handler satisfies an uncaught exception. |
| • exceptionType : Classsifier [1..*] | The kind of instances that the handler catches. If an exception occurs whose type is any of the classifiers in the set, the handler catches the exception and executes its body. |
| • exceptionInput : ObjectNode [1..1] | An object node within the handler body. When the handler catches an exception, the exception token is placed in this node, causing the body to execute. |

### Constraints

[1] The exception body may not have any explicit input or output edges.

[2] The result pins of the exception handler body must correspond in number and types to the result pins of the protected node.

[3] The handler body has one input, and that input is the same as the exception input.

### Semantics

If a RaiseExceptionAction is executed, all the tokens in the immediately containing structured node or activity are terminated. Then the set of execution handlers on the structured node or invocation action of the activity is examined for a handler that matches the exception. A handler matches if the type of the exception is the same as or a descendant of one of the exception classifiers specified in the handler. If there is a match, the handler "catches" the exception. The exception object is placed in the exceptionInput node as a token to start execution of the handler body.

If the exception is not caught by any of the handlers on the node or invocation action, the exception handling process repeats, propagating to the enclosing structured node or activity. If the exception propagates to the topmost level of the system and is not caught, the behavior of the system is unspecified. Profiles may specify what happens in such cases.

The handler body has no explicit input or output edges. It has the same access to its surrounding context as the protected node. The result tokens of the handler body become the result tokens of the protected node. Any control edges leaving the protected node receive control tokens on completion of execution of the handler body with the handler catching the exception. When the handler body completes execution, it is as if the protected node had completed execution.

### Notation

The notation for exception handlers is illustrated in Figure 12.79. An exception handler for a protected node is shown by drawing a "lightning bolt" symbol from the boundary of the protected node to a small square on the boundary of the exception handler. The name of the exception type is placed next to the lightning bolt. The small square is the exception input node, and it must be owned by the handler body. Its type is the given exception type. Both the protected node and the exception handler must be at the same nesting level. (Otherwise the notation could be misinterpreted as an interrupting edge, which crosses a boundary.) Multiple exception handlers may be attached to the same protected node, each by its own lightning bolt.

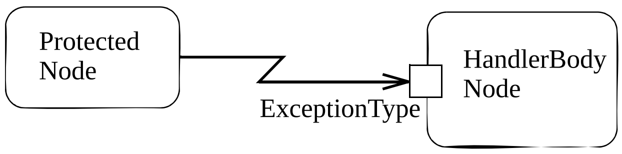


**Figure 12.79 - Exception Handler Notation**

### Presentation Options

An option for notating an interrupting edge is a zig-zag adornment on a straight line.

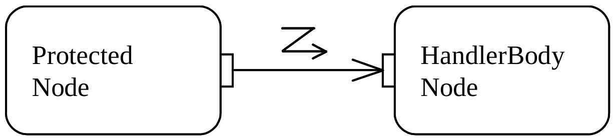


**Figure 12.80 - Exception Handler Presentation option**

### Examples

Figure 12.81 shows a matrix calculation. First a matrix is inverted, then it is multiplied by a vector to produce a vector. If the matrix is singular, the inversion will fail and a SingularMatrix exception occurs. This exception is handled by the exception handler labeled SingularMatrix, which executes the region containing the SubstituteVector1 action. If an overflow exception occurs during either the matrix inversion or the vector multiplication, the region containing the SubstituteVector2 action is executed.

The successors to an exception handler body are the same as the successors to the protected node. It is unnecessary to show control flow from the handler body. Regardless of whether the matrix operations complete without exception or whether one of the exception handlers is triggered, the action PrintResults is executed next.

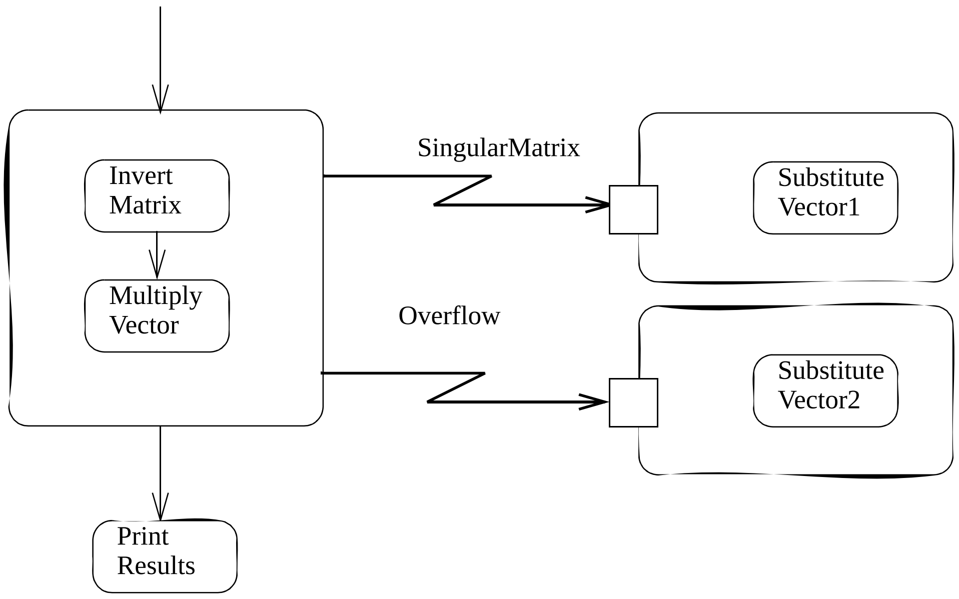


**Figure 12.81 - Exception Handler example**

### Changes from previous UML

ExceptionHandler replaces JumpHandler in UML 1.5.

Modeling of traditional break and continue statements can be accomplished using direct control flow from the statement to the control target. UML 1.5 combined the modeling of breaks and continues with exceptions, but that is no longer necessary and it is not recommended in this specification.

### 12.3.24 ExecutableNode (from ExtraStructuredActivities, StructuredActivities)

#### Generalizations

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

#### Description

An executable node is an abstract class for activity nodes that may be executed. It is used as an attachment point for exception handlers.

#### Associations

*Package ExtraStructuredActivities*

| | |
|---|---|
| • handler : ExceptionHandler [0..*] | A set of exception handlers that are examined if an uncaught exception propagates to the outer level of the executable node. |

### 12.3.25 ExpansionKind (from ExtraStructuredActivities)

#### Generalizations

None

#### Description

ExpansionKind is an enumeration type used to specify how multiple executions of an expansion region interact. See “ExpansionRegion (from ExtraStructuredActivities).”

#### Enumeration Literals

| | |
|---|---|
| • parallel | The executions are independent. They may be executed concurrently. |
| • iterative | The executions are dependent and must be executed one at a time, in order of the collection elements. |
| • stream | A stream of collection elements flows into a single execution, in order of the collection elements. |

### 12.3.26 ExpansionNode (from ExtraStructuredActivities)

#### Generalizations

- “ObjectNode (from BasicActivities, CompleteActivities)” on page 380

#### Description

An expansion node is an object node used to indicate a flow across the boundary of an expansion region. A flow into a region contains a collection that is broken into its individual elements inside the region, which is executed once per element. A flow out of a region combines individual elements into a collection for use outside the region.

### Associations

| | |
|---|---|
| • regionAsInput : ExpansionRegion[0..1] | The expansion region for which the node is an input. |
| • regionAsOutput : ExpansionRegion[0..1] | The expansion region for which the node is an output. |

### Semantics

See “ExpansionRegion (from ExtraStructuredActivities).”

### Notation

See “ExpansionRegion (from ExtraStructuredActivities).”

## 12.3.27 ExpansionRegion (from ExtraStructuredActivities)

An expansion region is a structured activity region that executes multiple times corresponding to elements of an input collection.

### Generalizations

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

### Description

An expansion region is a strictly nested region of an activity with explicit input and outputs (modeled as ExpansionNodes). Each input is a collection of values. If there are multiple input pins, each of them must hold the same kind of collection, although the types of the elements in the different collections may vary. The expansion region is executed once for each element (or position) in the input collection.

If an expansion region has outputs, they must be collections of the same kind and must contain elements of the same type as the corresponding inputs. The number of output collections at runtime can differ from the number of input collections. On each execution of the region, an output value from the region is inserted into an output collection at the same position as the input elements. If the region execution ends with no output, then nothing is added to the output collection. When this happens the output collection will not have the same number of elements as the input collections, the region acts as a filter. If all the executions provide an output to the collection, then the output collections will have the same number of elements as the input collections.

The inputs and outputs to an expansion region are modeled as ExpansionNodes. From “outside” of the region, the values on these nodes appear as collections. From “inside” the region the values appear as elements of the collections. Object flow edges connect pins outside the region to input and output expansion nodes as collections. Object flow edges connect pins inside the region to input and output expansion nodes as individual elements. From the inside of the region, these nodes are visible as individual values. If an expansion node has a name, it is the name of the individual element within the region.

Any object flow edges that cross the boundary of the region, without passing through expansion nodes, provide values that are fixed within the different executions of the region.

### Attributes

- mode : ExpansionKind - The way in which the executions interact:
  parallel - all interactions are independent.
  iterative - the interactions occur in order of the elements.
  stream - a stream of values flows into a single execution.

## Associations

- inputElement : ExpansionNode[1..*]
  An object node that holds a separate element of the input collection during each of the multiple executions of the region.
- outputElement : ExpansionNode[0..*]
  An object node that accepts a separate element of the output collection during each of the multiple executions of the region. The values are formed into a collection that is available when the execution of the region is complete.

## Constraints

[1] An ExpansionRegion must have one or more argument ExpansionNodes and zero or more result ExpansionNodes.

## Semantics

When an execution of an activity makes a token available to the input of an expansion region, the expansion region consumes the token and begins execution. The expansion region is executed once for each element in the collection (or once per element position, if there are multiple collections). The concurrency attribute controls how the multiple executions proceed:

- If the value is *parallel*, the execution may happen in parallel, or overlapping in time, but they are not required to.
- If the value is *iterative*, the executions of the region must happen in sequence, with one finishing before another can begin. The first iteration begins immediately. Subsequent iterations start when the previous iteration is completed. During each of these cases, one element of the collection is made available to the execution of the region as a token during each execution of the region. If the collection is ordered, the elements will be presented to the region in order; if the collection is unordered, the order of presenting elements is undefined and not necessarily repeatable. On each execution of the region, an output value from the region is inserted into an output collection at the same position as the input elements.
- If the value is *stream*, there is a single execution of the region, but its input place receives a stream of elements from the collection. The values in the input collection are extracted and placed into the execution of the expansion region as a stream in order, if the collection is ordered. Such a region must handle streams properly or it is ill defined. When the execution of the entire stream is complete, any output streams are assembled into collections of the same kinds as the inputs.

## Notation

An expansion region is shown as a dashed rounded box with one of the keywords *parallel*, *iterative*, or *streaming* in the upper left corner.

Input and output expansion nodes are drawn as small rectangles divided by vertical bars into small compartments. (The symbol is meant to suggest a list of elements.) The expansion node symbols are placed on the boundary of the dashed box. Usually arrows inside and outside the expansion region will distinguish input and output expansion nodes. If not, then a small arrow can be used as with Pins (see Figure 12.124 on page 390).

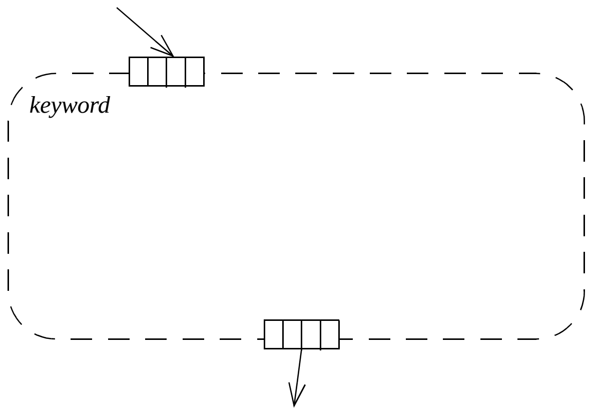


**Figure 12.82 - Expansion region**

As a shorthand notation, the “list box pin” notation may be placed directly on an action symbol, replacing the pins of the action (Figure 12.83). This indicates an expansion region containing a single action. The equivalent full form is shown in Figure 12.84.

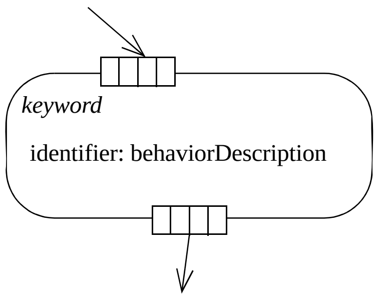


**Figure 12.83 - Shorthand notation for expansion region containing single node**

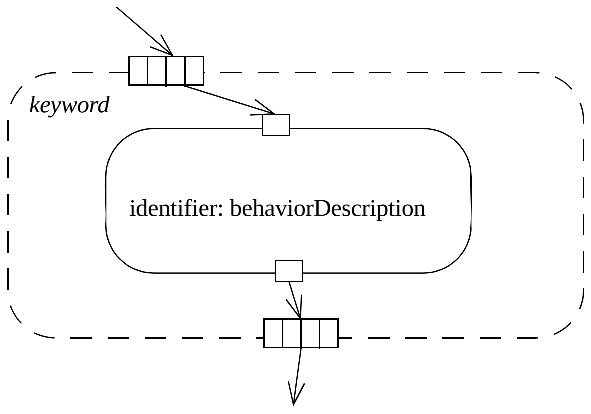


**Figure 12.84 - Full form of previous shorthand notation**

## Presentation Options

The notation in Figure 12.85 maps to an expansion region in parallel mode, with one behavior invoked in the region, as shown below.

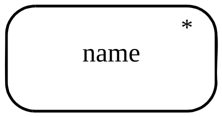


**Figure 12.85 - Notation for expansion region with one behavior invocation**

## Examples

Figure 12.86 shows an expansion region with two inputs and one output that is executed in parallel. Execution of the region does not begin until both input collections are available. Both collections must have the same number of elements. The interior activity fragment is executed once for each position in the input collections. During each execution of the region, a pair of values, one from each collection, is available to the region on the expansion nodes. Each execution of the region produces a result value on the output expansion node. All of the result values are formed into a collection of the same size as the input collections. This output collection is available outside the region on the result node after all the parallel executions of the region have completed.

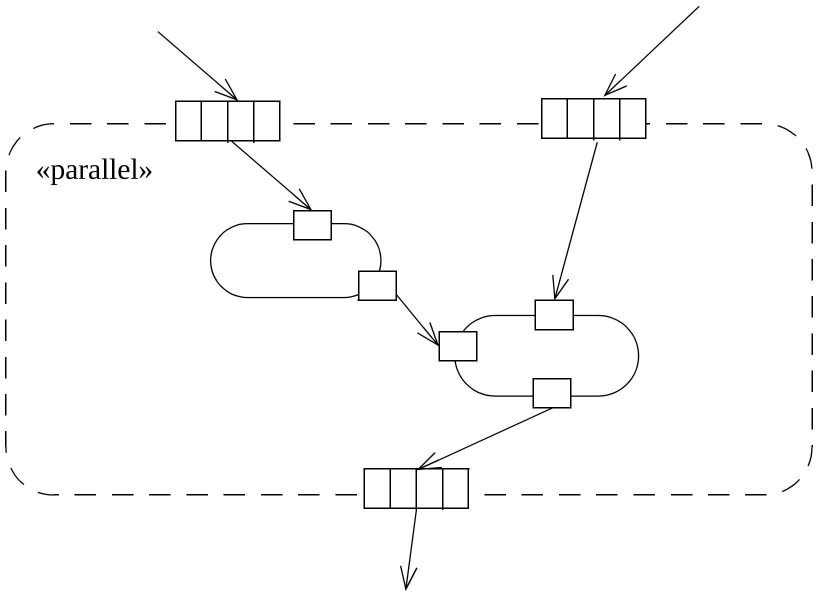


**Figure 12.86 - Expansion region with 2 inputs and 1 output**

Figure 12.86 shows a fragment of a Fast Fourier Transform (FFT) computation containing an expansion region. Outside the region, there are operations on arrays of complex numbers. S, Slower, Supper, and V are arrays. Cut and shuffle are operations on arrays. Inside the region, two arithmetic operations are performed on elements of the 3 input arrays, yielding 2 output arrays. Different positions in the arrays do not interact, therefore the region can be executed in parallel on all positions.

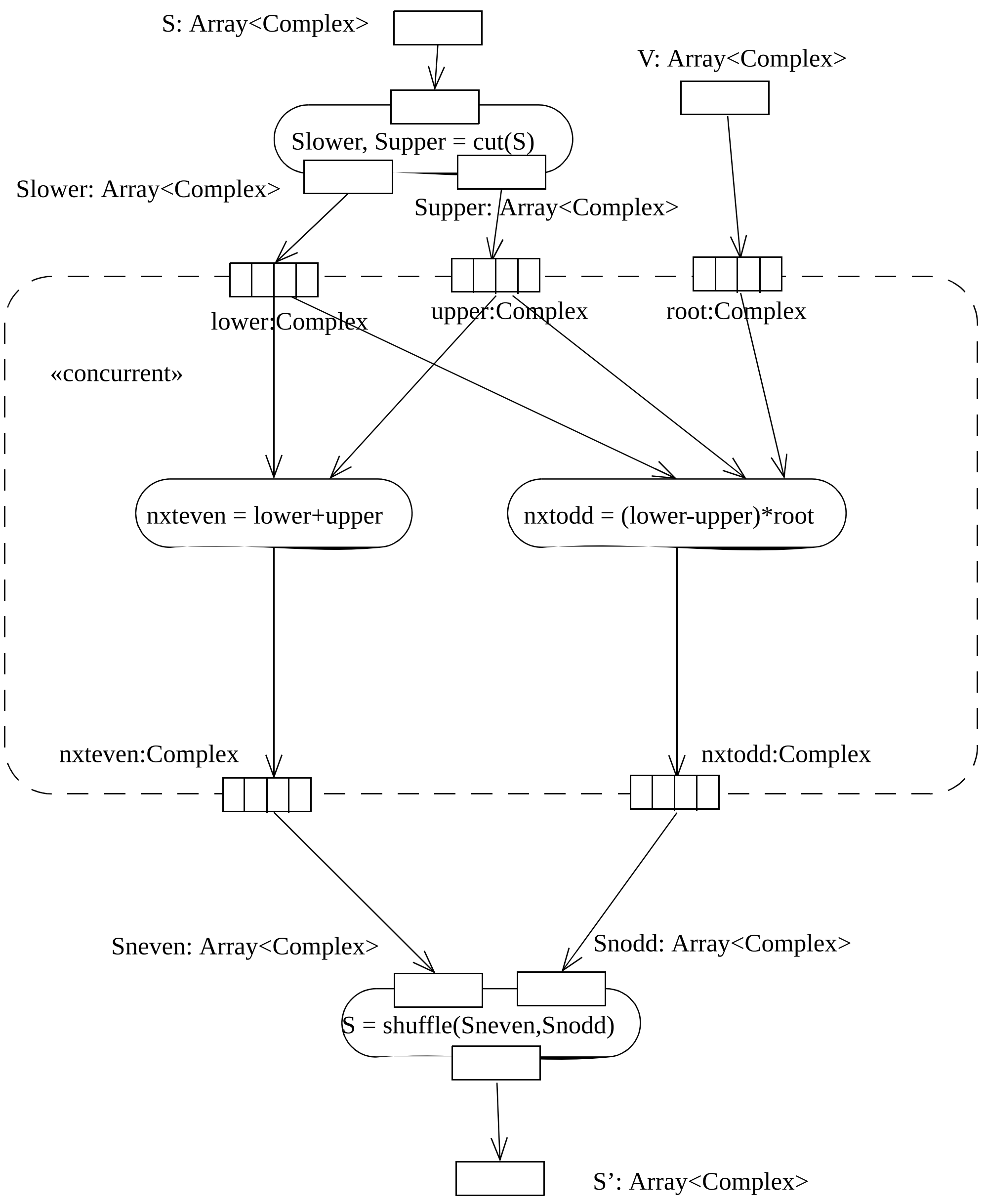


**Figure 12.87 - Expansion region**

The following example shows a use of the shorthand notation for an expansion region with a single action. In this example, the trip route outputs sets of flights and sets of hotels to book. The hotels may be booked independently and in parallel with each other and with booking the flight.

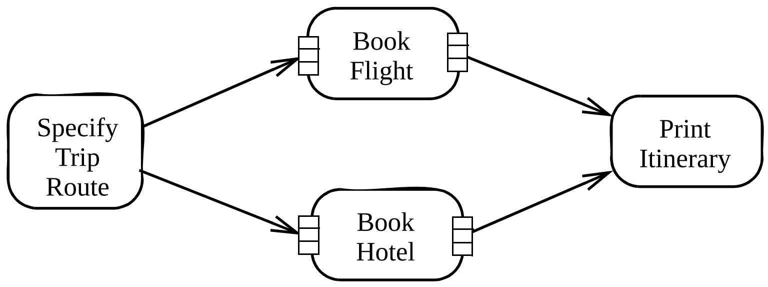


**Figure 12.88 -Examples of expansion region shorthand**

Specify Trip Route below can result in multiple flight segments, each of which must be booked separately. The Book Flight action will invoke the Book Flight behavior multiple times, once for each flight segment in the set passed to BookFlight.

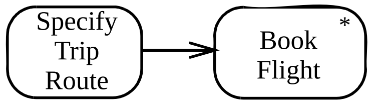


**Figure 12.89 - Shorthand notation for expansion region**

### Rationale

Expansion regions are introduced to support applying behaviors to elements of a set without constraining the order of application.

### Changes from previous UML

ExpansionRegion replaces MapAction, FilterAction, and dynamicConcurrency and dynamicMultiplicity attributes on ActionState. Dynamic multiplicities less than unlimited are not supported in UML 1.5.

## 12.3.28 FinalNode (from IntermediateActivities)

A final node is an abstract control node at which a flow in an activity stops.

### Generalizations

- "ControlNode (from BasicActivities)" on page 346

### Description

See descriptions at children of final node.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

[1] A final node has no outgoing edges.

### Semantics

All tokens offered on incoming edges are accepted. See children of final node for other semantics.

### Notation

The notations for final node are illustrated below. There are two kinds of final node: activity final and (IntermediateActivities) flow final. For more details on each of these specializations, see ActivityFinal and FlowFinal.

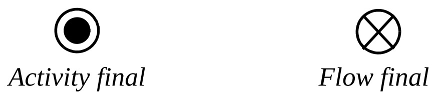


**Figure 12.90 - Final node notation**

### Examples

The figure below illustrates two kinds of final node: flow final and activity final. In this example, it is assumed that many components can be built and installed before finally delivering the resulting application. Here, the Build Component behavior occurs iteratively for each component. When the last component is built, the end of the building iteration is indicated with a flow final. However, even though all component building has come to an end, other behaviors are still executing. When the last component has been installed, the application is delivered. When Deliver Application has completed, control is passed to an activity final node—indicating that all processing in the activity is terminated.

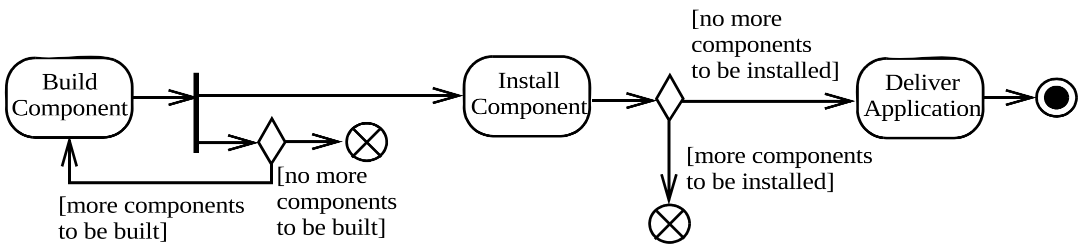

Build Component
Install Component
[no more components to be installed]
Deliver Application
[more components to be installed]
[no more components to be built]
[more components to be built]

**Figure 12.91 - Flow final and activity final example.**

### Rationale

Final nodes are introduced to model where flows end in an activity.

### Changes from previous UML

FinalNode replaces the use of FinalState in UML 1.5 activity modeling, but its concrete classes have different semantics than FinalState.

## 12.3.29 FlowFinalNode (from IntermediateActivities)

A flow final node is a final node that terminates a flow.

### Generalizations

- "FinalNode (from IntermediateActivities)" on page 360

### Description

A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

Flow final destroys tokens flowing into it.

### Notation

The notation for flow final is illustrated below.

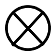


**Figure 12.92 - Flow final notation**

### Examples

In the example below, it is assumed that many components can be built and installed. Here, the Build Component behavior occurs iteratively for each component. When the last component is built, the end of the building iteration is indicated with a flow final. However, even though all component building has come to an end, other behaviors are still executing (such as Install Component).

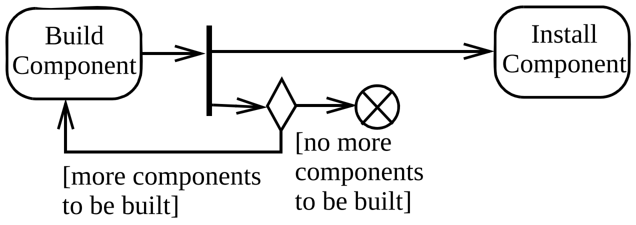

Build Component
Install Component
[no more components to be built]
[more components to be built]

**Figure 12.93 - Flow final example without merge edge**

### Rationale

Flow final nodes are introduced to model termination of a flow in an activity.

### Changes from previous UML

Flow final is new in UML 2.0.

## 12.3.30 ForkNode (from IntermediateActivities)

A fork node is a control node that splits a flow into multiple concurrent flows.

### Generalizations

- "ControlNode (from BasicActivities)" on page 346

### Description

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

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

[1] A fork node has one incoming edge.

[2] The edges coming into and out of a fork node must be either all object flows or all control flows.

### Semantics

Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target. The rest of the outgoing edges do not receive a token (these are the ones with failing guards). This is an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream (see

“Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” on page 306). When an offered token is accepted on all the outgoing edges, duplicates of the token are made and one copy traverses each edge. No duplication is necessary if there is only one outgoing edge, but it is not a useful case.

If guards are used on edges outgoing from forks, the modelers should ensure that no downstream joins depend on the arrival of tokens passing through the guarded edge. If that cannot be avoided, then a decision node should be introduced to have the guard, and shunt the token to the downstream join if the guard fails. See example in Figure 12.44 on page 319.

### Notation

The notation for a fork node is simply a line segment, as illustrated on the left side of the figure below. In usage, however, the fork node must have a single activity edge entering it, and two or more edges 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 RFP supports the interchange of diagram elements and their mapping to model elements.

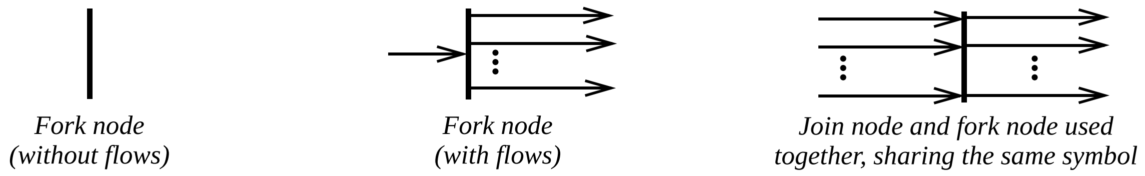

*Fork node*
*(without flows)*
*Fork node*
*(with flows)*
*Join node and fork node used*
*together, sharing the same symbol*

**Figure 12.94 - Fork node notation**

### Examples

In the example below, the fork node passes control to both the Ship Order and Send Invoice behaviors when Fill Order is completed.

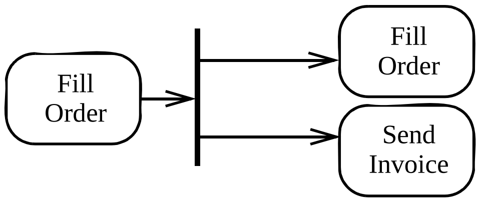


**Figure 12.95 - Fork node example.**

### Rationale

Fork nodes are introduced to support parallelism in activities.

### Changes from previous UML

Fork nodes replace the use of PseudoState with fork kind in UML 1.5 activity modeling. State machine forks in UML 1.5 required synchronization between parallel flows through the state machine RTC step. UML 2.0 activity forks model unrestricted parallelism.

### 12.3.31 InitialNode (from BasicActivities)

An initial node is a control node at which flow starts when the activity is invoked.

#### Generalizations

- "ControlNode (from BasicActivities)" on page 346

#### Description

An activity may have more than one initial node.

#### Attributes

No additional attributes

#### Associations

No additional associations

#### Constraints

[1] An initial node has no incoming edges.

[2] Only control edges can have initial nodes as source.

#### Semantics

An initial node is a starting point for executing an activity (or structured node, see "StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)" on page 396). A control token is placed at the initial node when the activity starts, but not in initial nodes in structured nodes contained by the activity. Tokens in an initial node are offered to all outgoing edges. If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node. For convenience, initial nodes are an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream, for example, by guards (see Activity). This is equivalent to interposing a CentralBufferNode between the initial node and its outgoing edges.

Note that flows can also start at other nodes, see ActivityParameterNode and AcceptEventAction, so initial nodes are not required for an activity to start execution. In addition, when an activity starts, control tokens are placed at actions and structured nodes that have no incoming edges, except if they are handler bodies (see "ExceptionHandler (from ExtraStructuredActivities)" on page 351) are fromActions of action input pins, or are contained in structured nodes.

#### Notation

Initial nodes are notated as a solid circle, as indicated in the figure below.

**Figure 12.96 - Initial node notation**

### Examples

In the example below, the initial node passes control to the Receive Order behavior at the start of an activity.

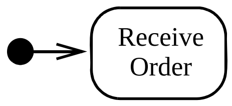


**Figure 12.97 - Initial node example**

### Rationale

Initial nodes are introduced to model where flows start in an activity.

### Changes from previous UML

InitialNode replaces the use of PseudoState with kind initial in UML 1.5 activity modeling.

## 12.3.32 InputPin (as specialized)

Input pins are object nodes that receive values from other actions through object flows. See Pin, Action, and ObjectNode for more details.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

[1] Input pins may only have outgoing edges when they are on actions that are structured nodes, and these edges must target a node contained by the structured node.

### Semantics

See "InputPin (from BasicActions)" on page 249

## 12.3.33 InterruptibleActivityRegion (from CompleteActivities)

An interruptible activity region is an activity group that supports termination of tokens flowing in the portions of an activity.

### Generalizations

- "ActivityGroup (from BasicActivities, FundamentalActivities)" on page 322

### Description

An interruptible region contains activity nodes. When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviors in the region are terminated.

### Attributes

No additional attributes

### Associations

- interruptingEdge : ActivityEdge [0..*]    The edges leaving the region that will abort other tokens flowing in the region.

### Constraints

[1] Interrupting edges of a region must have their source node in the region and their target node outside the region in the same activity containing the region.

### Semantics

The region is interrupted, including accept event actions in the region, when a token traverses an interrupting edge. At this point the interrupting token has left the region and is not terminated. AcceptEventActions in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the accept event action.

Token transfer is still atomic, even when using interrupting regions. If a non-interrupting edge is passing a token from a source node in the region to target node outside the region, then the transfer is completed and the token arrives at the target even if an interruption occurs during the traversal. In other words, a token transition is never partial; it is either complete or it does not happen at all.

Do not use an interrupting region if it is not desired to abort all flows in the region in some cases. 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 leaves the region. Arrange for separate invocations of the activity to use separate executions of the activity when employing interruptible regions, so tokens from each invocation will not affect each other.

### Notation

An interruptible activity region is notated by a dashed, round-cornered rectangle drawn around the nodes contained by the region. An interrupting edge is notation with a lightning-bolt activity edge.

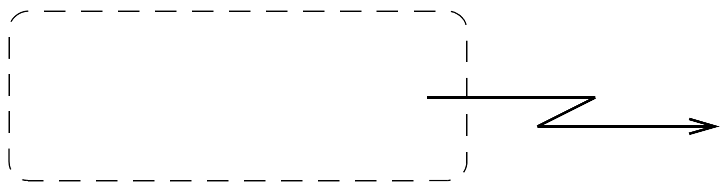


**Figure 12.98 - InterruptibleActivityRegion notation with interrupting edge**

### Presentation Options

An option for notating an interrupting edge is a zig zag adornment on a straight line.

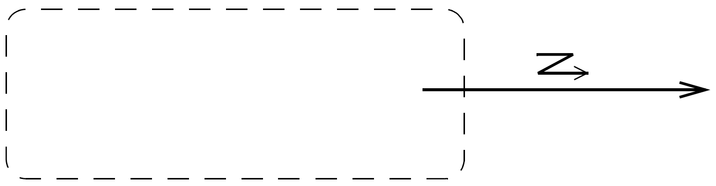


**Figure 12.99 - InterruptibleActivityRegion notation with interrupting edge**

### Examples

The first figure below illustrates that when an order cancellation request is made—only while receiving, filling, or shipping) orders—the Cancel Order behavior is invoked.

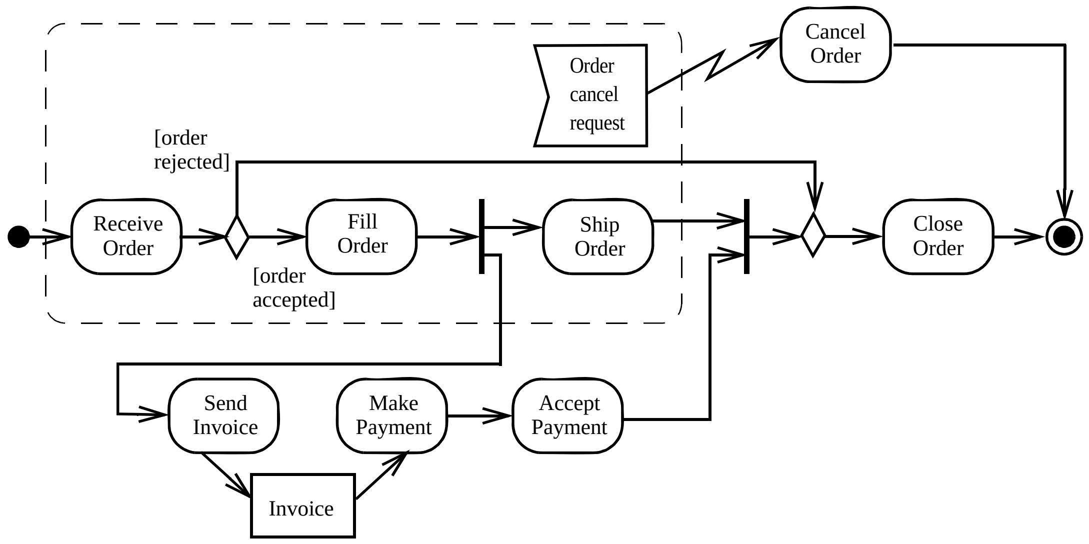


**Figure 12.100 - InterruptibleActivityRegion example**

### Rationale

Interruptible regions are introduced to support more flexible non-local termination of flow.

### Changes from previous UML

Interruptible regions in activity modeling are new to UML 2.0.

## 12.3.34 JoinNode (from CompleteActivities, IntermediateActivities)

A join node is a control node that synchronizes multiple flows.