

## UML 2 Activity and Action Models

### Part 6: Structured Activities

**Conrad Bock**, U.S. National Institute of Standards and Technology

This is the sixth in a series introducing the activity model in the Unified Modeling Language, version 2 (UML 2), and how it integrates with the action model [1]. The first article gives an overview of activities and actions [2], while the next four cover models typically used for graphical flow languages. This one describes models for languages that usually have textual presentations. It covers structured nodes for sequencing, conditionals, loops, and expansion regions for operating on collections, as well as exception handlers, variables, and action pins.

### 1 STRUCTURED ACTIVITY MODELS

Two of the most common process notation styles are:

- Graphical notations usually support less restricted patterns of control flow. They normally use data flow rather than variables to pass data between actions.
- Textual notations are usually designed for well-nested control, even if they support non-local control flow in exceptional cases. They typically use variables to pass data between actions, rather than data flow.

Ideally, there would be a single underlying model for these,<sup>1</sup> but to simplify translation from notation to repository, UML activities have models for both presentation styles, informally called flow and structure models. Unfortunately, the models are not independent, because the structured elements mainly address control, still requiring flow

---

<sup>1</sup> Structured and flow models are equivalent as long as the value of each variable in the structured model is only set once, and queuing is not used in the flow model. With a single underlying model, the various notations could be easily compared and integrated. However, in general this requires complicated translations between notation and repository. It may be easy enough when diagrams are translated all at once, but more difficult when translated incrementally as the modeler edits the notation. Mappings from notation to model also affect monitoring and debugging execution at the level of the original notation. In the long run it can be expected that model compilation and execution visualization will become as sophisticated as they are in convention languages, and translations between many notations and a single underlying model more routine.

elements to pass data to actions in most cases (see example in). This is primarily because the structured models are based on those provided in UML 1.5, which uses flows for passing data between actions, and control flow for sequencing actions. Once the independence of structured and flow models is achieved, the two models can be applied in stages. For example, a textual language compiler could begin by translating a structured notation to the structured model, and then transform that to a flow model for data flow analysis.<sup>2</sup>

The dependencies of structured and flow packages are shown in Figure 1.<sup>3</sup> The two parts have a small common base at `FUNDAMENTALACTIVITIES`, which defines activities as having activity nodes, without control or data flow. The flow models, starting with `BASICACTIVITIES`, are independent of the structured models, and introduce control and object flow edges along with other flow-based elements. The basic structured model in `STRUCTUREDACTIVITIES` is independent of flows, and supports structured control constructs rather than control flow, and variables instead of data flow. Structured models beyond this, in `COMPLETEACTIVITIES` and `EXTRASTRUCTUREDACTIVITIES`, combine structured and flow models, for the reasons given above.

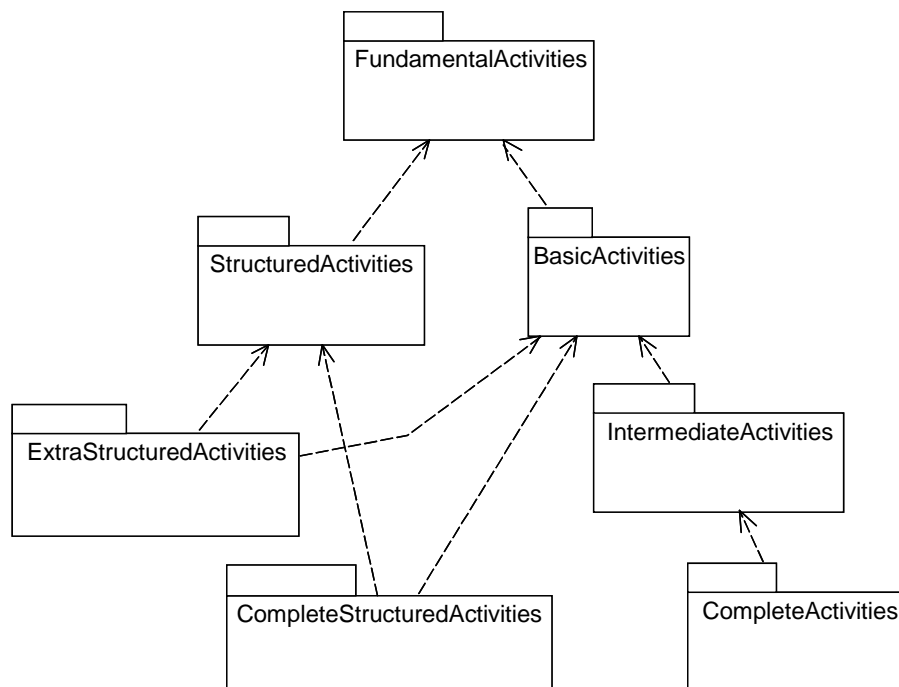
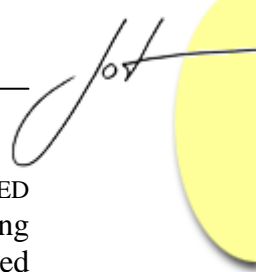


Figure 1: Activities Package Dependency

<sup>2</sup> The UML repository also supports tabular or matrix formats such as the Dependency Structure Matrix [3]. These provide a compact way to show function dependencies, by omitting some control information, but are not restricted to hierarchical decomposition.

<sup>3</sup> Compare to packaging before finalization in Figure 2 of [2].



The structured models in `STRUCTUREDACTIVITIES` and `COMPLETESTRUCTUREDACTIVITIES` have elements analogous to control constructs found in typical programming languages, although they are more general. They are kinds of activity nodes called *structured activity nodes*, with specializations for sequencing, conditionals, loops, and expansion regions for operating on collections, as described in sections 2 through 6. The first three are familiar from conventional programming languages, while the expansion regions are for operating on elements of a collection. The structured models are more general than typical programming languages, providing inputs and outputs for control constructs, as well as parallelism and pipelining.

Activities have structured and flow models for throwing and catching exceptions. The model in `EXTRASTRUCTUREDACTIVITIES` supports protecting structured nodes and actions, and catching exceptions thrown by the `RAISEEXCEPTIONACTION`, described in section 7. It is more general than typical programming constructs, for example by providing outputs from exception handlers. The model in `COMPLETEACTIVITIES` defines two flow-oriented constructs for exceptions, interruptible regions and exception parameters. These are covered as part of exception handling, even though they are not part of the structured models.

The model in `COMPLETESTRUCTUREDACTIVITIES` provides for a “pull” style of data flow, to complement the “push” style available in the flow model. In the pull style an action starts when another needs a value from it, whereas in the push style an action starts when another provides values to it. The pull style is applicable to modeling nested expressions in programming languages, as covered in section 8.

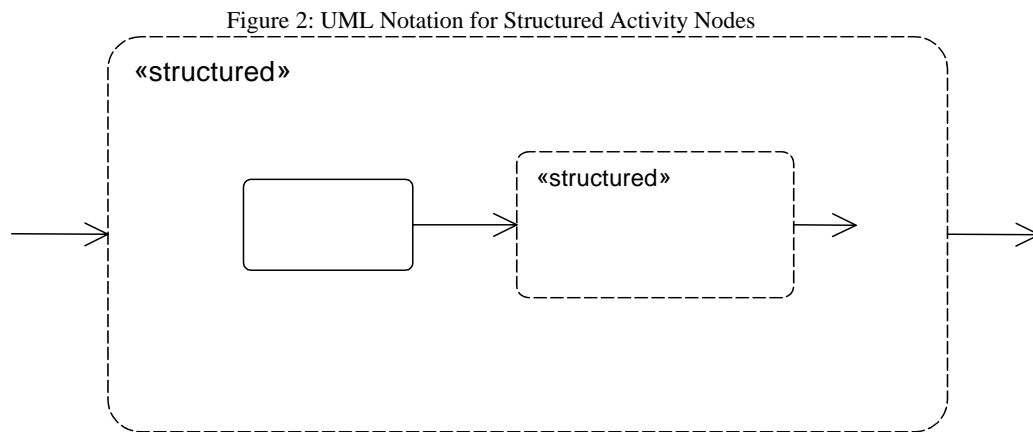
Most of the structured models do not have a standard UML notation, because they are intended for generation from various programming and action languages.<sup>4</sup> Examples in the following sections use the textual syntax of typical programming languages, but only as a visual presentation of the models, borrowing their keywords and punctuation without their implementations [4]. Repository models are given to show how the notations translate to instances of the UML metamodel.

---

<sup>4</sup> See Appendix B of UML 1.5 [5] for example action languages.

## 2 STRUCTURED ACTIVITY NODES IN GENERAL

Structured activity nodes are nodes that contain other nodes, the only kind of activity node that does. A node cannot be directly contained by more than one structured node. Structured nodes may contain other structured nodes, so an action may be indirectly contained by many structured nodes, but it will have only one that immediately contains it. Figure 2 shows the standard UML notation for structured nodes, with an example of a nested structured node and action. Structured nodes are a natural model for blocks in textual languages, since opening and closing delimiters, such as parentheses and braces, always match each other in an unambiguous and well-nested way.



Structured activity nodes do not enforce well-nested flows. For example, an action can provide and accept control and data to and from actions outside the structured node at any point in the execution of the node. Textual languages allow this also, with “go to” commands for example, though methodologies normally discourage it. These methodologies can be applied to structured nodes if desired.

Like all nodes, a structured node can participate in control flows, and be contained in structured control constructs like conditionals and loops (sections 3 through 6). Complete structured nodes can also have pins and participate in object flows [2][6]. When a structured node has all its required incoming control and data, it starts the nodes in it according to the same rules that an activity starts its nodes. Directly contained initial nodes, actions, and structured nodes that do not have incoming edges will start when their containing node does.<sup>5</sup> Conversely, nodes in a structured node cannot start unless the structured node does, and can execute only as long as the structured node does. When control in a structured node reaches a directly contained activity final node, the structured node and its contents are terminated according to the same rules used for activity final nodes directly contained by activities [7]. When a structured node completes, its outgoing control edges receive control. If the node has output pins, the outgoing data edges receive the values in the pins, or a null token if there are no values.

Structured nodes may also declare variables. For example, the code fragment shown in

<sup>5</sup> The rules for starting nodes in activities and structured nodes were clarified in the finalized UML 2 [1].

Figure 3 is an example nonstandard notation for the UML repository model in Figure 4. Curly braces are used to notate a structured node. The variable declaration appears in the repository model as a link between the structured node and a variable. The initialization is modeled with one of the actions for modifying variable values, `ADDVARIABLEVALUEACTION`. The initial value is specified by a kind of input pin that determines the input value by evaluating a value specification, `VALUEPIN`, see section 8. Variables can also be declared on activities in a similar way. The complete set of actions on variables will be described in a later article.

```
{
  int count = 1;
}
```

Figure 3: Example Textual Notation for Variables

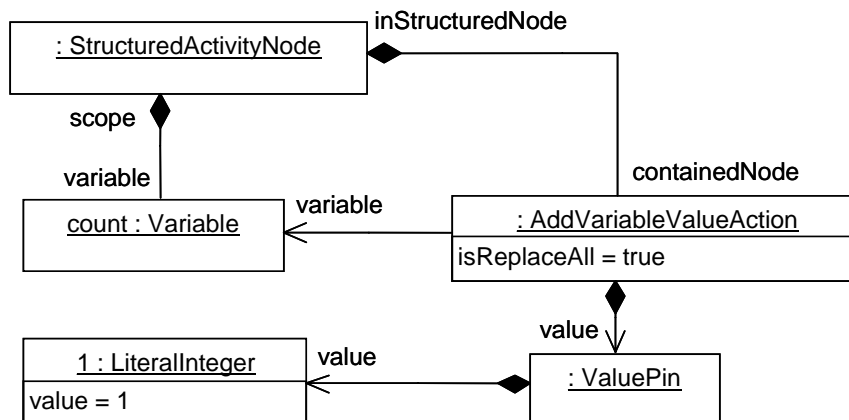


Figure 4: Repository Model for Figure 3

Structured nodes have an option to model what is usually referred to as a transaction. If the Boolean property `MUSTISOLATE` is true for a structured node, then any object used by an action within the node cannot be accessed in a conflicting way by any action outside the node until the node as a whole completes.<sup>6</sup> Isolation does not imply that rollback (atomicity) or other transaction mechanisms must be used to satisfy it, though they can be. It can be notated graphically on structured nodes with the UML property notation `{ mustIsolate = true }`.

<sup>6</sup> The current specification could be clearer that this is a requirement on the execution of a structured node, which is why the prefix is “must” rather than “is” (compare `ISASSURED` and `ISDETERMINANT` in section 4).

### 3 SEQUENCE NODE

The most basic structured node executes a series of actions. An example notation taken from C [8] is shown in Figure 5, omitting variables and parameters for brevity. The corresponding UML repository model in Figure 6. The sequence node contains three call behavior actions in order. The notation “{1}” in the repository model is not standard UML, but indicates the order of links of the EXECUTABLENODE association from the sequence node. Executable nodes are a class of nodes in the UML metamodel that include actions and structured nodes.<sup>7</sup>

```
{
  CheckOrder();
  FillOrder();
  CloseOrder();
}
```

Figure 5: Example Textual Notation for Sequences

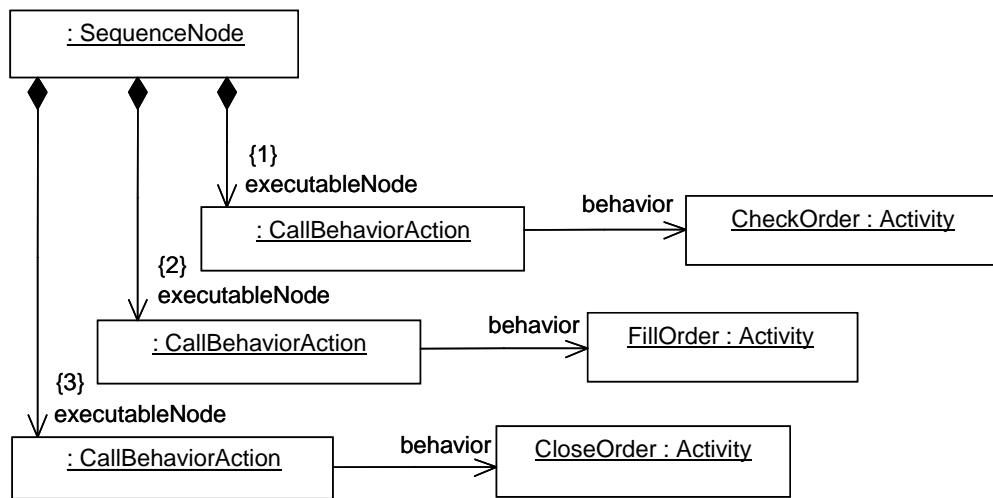


Figure 6: Repository Model for Figure 5

As mentioned in the previous section, the structured models still depend on flows to model most of what is done with variables and parameters in textual languages. For example, Figure 7 adds input parameters to specify which order will be checked, filled, and closed. The corresponding flow model in uses object flows from activity parameter nodes [2][6] to pass an order from the input parameter to the actions, and control flow

<sup>7</sup> Sequence nodes can take inputs and provide outputs, but do not currently identify output pins in the sequence that provide values to the output pins of the sequence, as conditionals and loops do. A workaround for the common case of sequences that have results calculated at the last step is to use object flows from output pins of actions in the sequence to the output pins of the sequence. This is useful for modeling some language constructs, such as Common LISP `progn` [9]. The general will be addressed in UML revision.

instead of a sequence node. The current version of UML does not have an action for accessing parameters as it does variables.

```

ProcessOrder(o : Order)
{
  CheckOrder(o);
  FillOrder(o);
  CloseOrder(o);
}

```

Figure 7: Example Textual Notation for Parameters

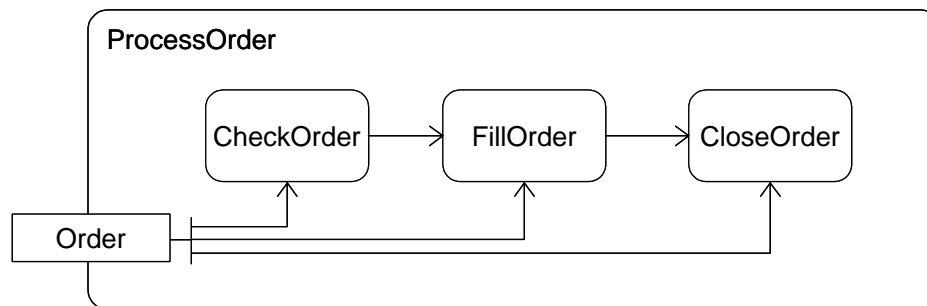


Figure 8: UML Notation for Parameters

## 4 CONDITIONAL NODE

The structured node for conditionals is composed of clauses that have test actions and body actions, where test actions determine which body actions are executed. An example textual notation taken from C is shown in Figure 9, for the repository model in Figure 10, omitting parameters and variables for brevity, as well as the containment links from the condition node the elements in the clauses.<sup>8</sup> The clauses in this example are completely ordered in execution, as specified by the precedence relation between them. The clause without a predecessor is the translation of the first “if” in the textual notation, and the one without a successor is the translation of the “else.” Each clause has a node for testing whether the clause succeeds and a node to execute if it does. Each clause identifies an output pin of the test action that provides a Boolean value determining if that clause succeeds, called the *decider* pin. The “else” clause has a test that always returns true, modeled with a value specification action. Test and body actions can be structured nodes, and the decider pin can be inside a test structured node. Test and body actions can also be sets of actions, where the first actions executed are the ones that do not have incoming edges.

<sup>8</sup> In UML 2 currently, test and body actions are not owned by clauses, and body output pins can be referred to by multiple clauses. In UML 1.5, clauses owned their test and body actions. The effect in UML 2 is that bodies can share actions. For example, the call to FILLORDER in Figure 10 could be shared between the first and second clauses. It is unclear if this is much of a benefit, since changing the body of one clause will change another, which may not be the intention.

```

{
  if CheckOrder() FillOrder();
  else if ModifyOrder() FillOrder();
  else CancelOrder();
}
    
```

Figure 9: Example Textual Notation for Conditionals

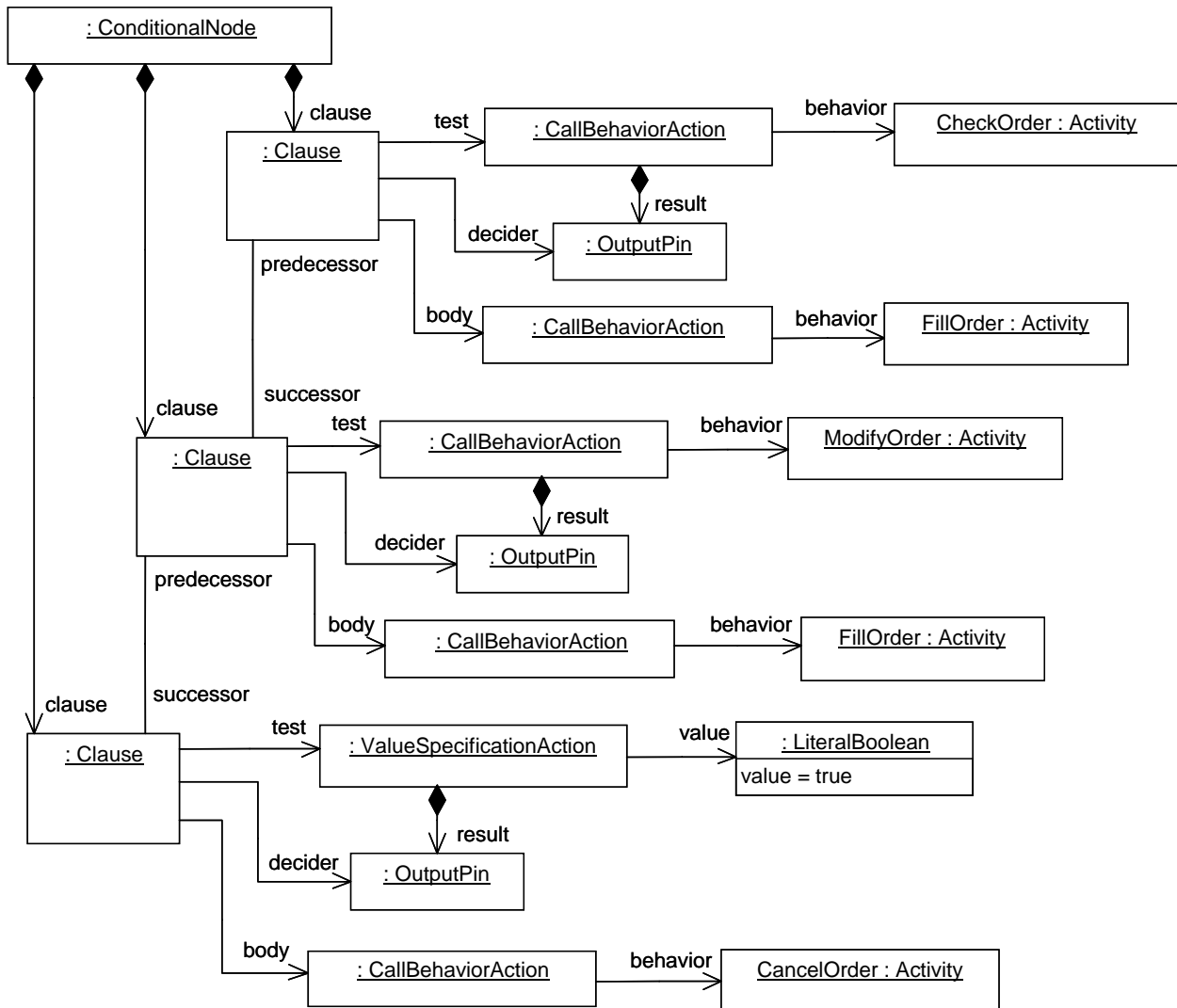
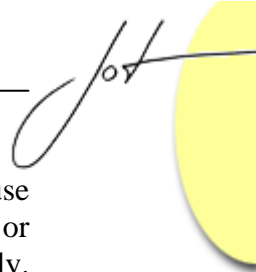


Figure 10: Repository Model for Figure 9



Conditional nodes are more general than typical programming conditionals, because clauses can be partially ordered with some clauses having the same predecessor, or multiple clauses with no predecessor.<sup>9</sup> This means some tests may proceed concurrently. If one test succeeds, the body of that clause is executed. If more than one test succeeds, only one clause will be executed, but UML does not define which it is. The specification does not require that all tests complete, or that the first one yielding true causes the other tests to be terminated, though implementation may choose to do this. For applications that require more specific semantics, conditionals can be marked with two Boolean properties for specifying the intended behavior of the conditional:

- ISASSURED = TRUE specifies that at least one test will succeed.
- ISDETERMINATE = TRUE specifies that at most one test will succeed.

These are properties a modeler declares to be true, rather than properties guaranteed to be true by the implementation. The modeler must ensure the activity is designed to meet these requirements if the property values are true. The implementation may optimize based on these properties, for example, by executing the body of the first succeeding test, and terminating other concurrent tests that are not done yet.

Conditional nodes can have output pins providing results to other actions. Output values are taken from output pins of the body of the succeeding clause, which are identified by the clause. This is applicable to languages that support conditional expressions. Figure 11 shows example textual notations taken from CommonLISP and C, with expressions that return the cost of the filled order. The additional repository elements to Figure 10 are shown in Figure 12. The BODYOUTPUT association identifies pins in each clause that will have their values copied to the outputs of the conditional when the clause succeeds. The number of body outputs on each clause must match the number of conditional outputs, and the types must be compatible.

```
(setq Cost
  (cond ((CheckOrder) (FillOrder))
        ((ModifyOrder) (FillOrder))
        (t (CancelOrder))))

Cost = ( CheckOrder() ? FillOrder()
        : ( ModifyOrder() ? FillOrder()
          : CancelOrder() ) )
```

Figure 11: Example Textual Notations for Conditionals with Outputs

<sup>9</sup> Conditional nodes are also more general than using decision and merge nodes [7] for the above reasons, and because decision nodes only route values based on the characteristics of each value.

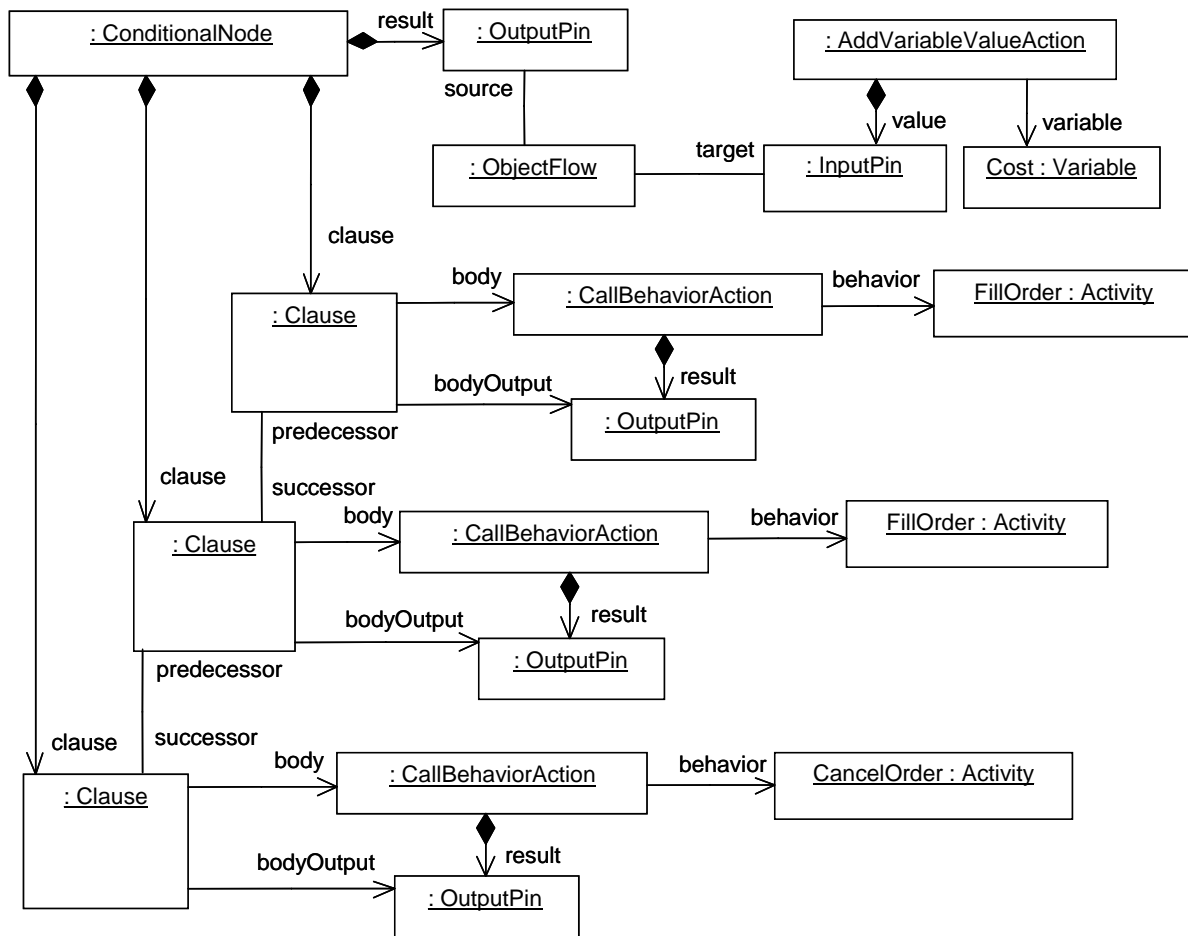
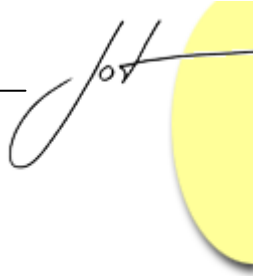


Figure 12: Repository Model for Figure 11

## 5 LOOPNODE

The structured node for loops is composed of setup actions, test actions, and body actions. Setup actions are performed once at the beginning of the loop, test actions are performed either at the beginning or end of each iteration to determine when to exit the loop, and the body actions are performed at each iteration.<sup>10</sup> Example textual notations taken from C are shown in Figure 13, for the repository model in Figure 14, omitting parameters and variables for brevity, as well as the containment links from the loop node to the elements in it. Loop nodes have an option to perform the test at beginning or the end of each iteration, as specified by the Boolean property `ISTESTEDFIRST`. As in conditionals, test and body actions in loops can be structured nodes, and the decider pin inside a test structured node. Test and body actions can also be sets of actions, where the

<sup>10</sup> Body actions are not organized into clauses as conditional actions are, because there would be only one clause.



first actions executed are the ones that do not have incoming edges.<sup>11</sup>

```

PrepareToProcessOrders ();
while (IsStockLeft ())
    ProcessOrder ();

for ( PrepareToProcessOrders () ; IsStockLeft () ; )
    ProcessOrder ();
    
```

Figure 13: Example Textual Notations for Loops

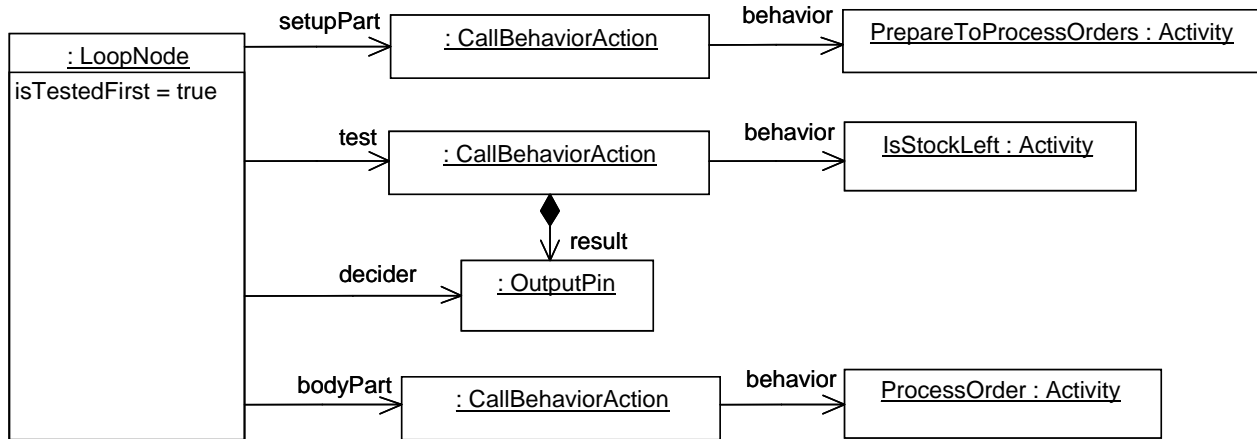


Figure 14: Repository Model for Figure 13

Loop nodes can have output pins providing results for input to other actions. Outputs are taken from pins called “loop variables,” which are not variables in the sense of Figure 4, but can be set at each iteration. These pins are initialized from input pins identified for that purpose. An example textual notation taken from CommonLISP is shown in Figure 15. Most of the additional repository elements to Figure 14 are shown in Figure 16.<sup>12</sup> The LOOPVARIABLE association specifies pins used for loop variables. In this example it is the one corresponding to AMOUNTSHIPPEDSO FAR in Figure 15.<sup>13</sup> The LOOPVARIABLEINPUT association specifies input pins that initialize loop variables when the loop node starts. The BODYOUTPUT associations identify output pins used to update the loop variables after each iteration. The values of the loop variables are moved to the output pins of the loop

<sup>11</sup> In UML 2 currently, test and body actions are not owned by loops. In UML 1.5, loops had a single clause, which owned its test and body actions.

<sup>12</sup> A complete model would include the BODYPART links to all the actions and pins on the flow from the call to PROCESSORDER to the body output pin.

<sup>13</sup> The current UML metamodel requires that output pins are owned by exactly one action through the OUTPUT association, or one of its specializations, which conflicts with pin ownership through the LOOPVARIABLE association. This will be addressed in revision.

when it is done.<sup>14</sup> Loop nodes can have input pins that are not loop variables. These provide constant values for the duration of the loop.

```
(setq TotalAmountShipped
  (progn (PrepareToProcessOrders)
    (do ((AmountShippedSoFar 0))
      ((IsStockLeft) AmountShippedSoFar)
      (setq AmountShippedSoFar
        (+ AmountShippedSoFar (ProcessOrder))))))
```

Figure 15: Example Textual Notation for Loops with Outputs

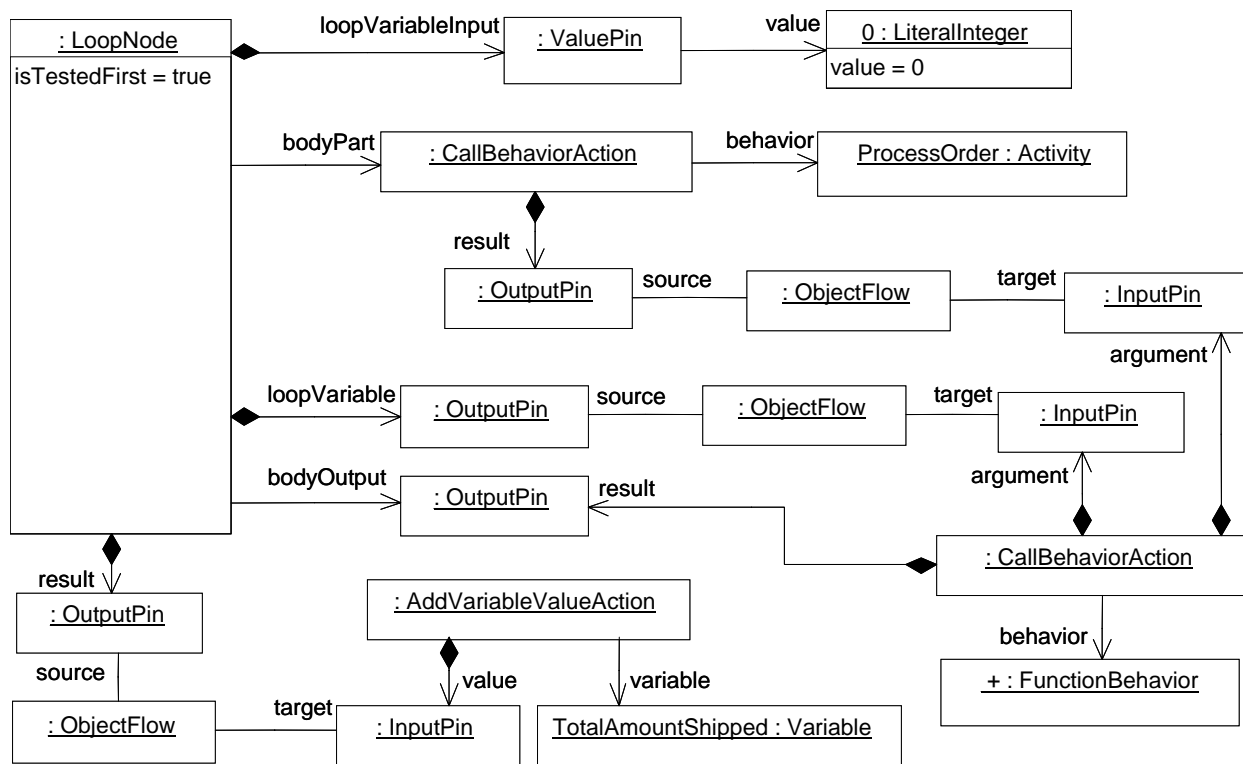


Figure 16: Repository Model for Figure 15

<sup>14</sup> Values remain in the loop variable pins unless they flow to inputs of actions in the body. This only occurs when all the required inputs to the action are available, including control [10], so the loop variables will still have their values when the test fails and the body does not start again.



## 6 EXPANSION REGION

An expansion region is a structured node that takes collections as input, acts on each element of the collections individually and produces elements to output collections. It can act on the elements iteratively, so that actions on each element proceed only after actions on the previous elements are done. It can also act on the elements in parallel, or as a pipelined stream of values through the nodes in the region. An example in standard UML notation is shown in Figure 17, with an example textual notation from CommonLISP in Figure 18. The input and output collections arrive and leave from specialized object nodes [10] called *expansion nodes*, which are notated as groups of small rectangles overlapping the boundary of the region. These have a similar function to pins, but have flows going in and coming out, and are not directly attached to actions. They have specialized semantics that transform collections to elements of the collection on input, and the reverse on output. The region outputs the filled orders only, so it filters the input collection (see [7] about decision nodes and decision input behaviors). Regions that produce an output for each input are equivalent to mappings over the collection.<sup>15</sup>

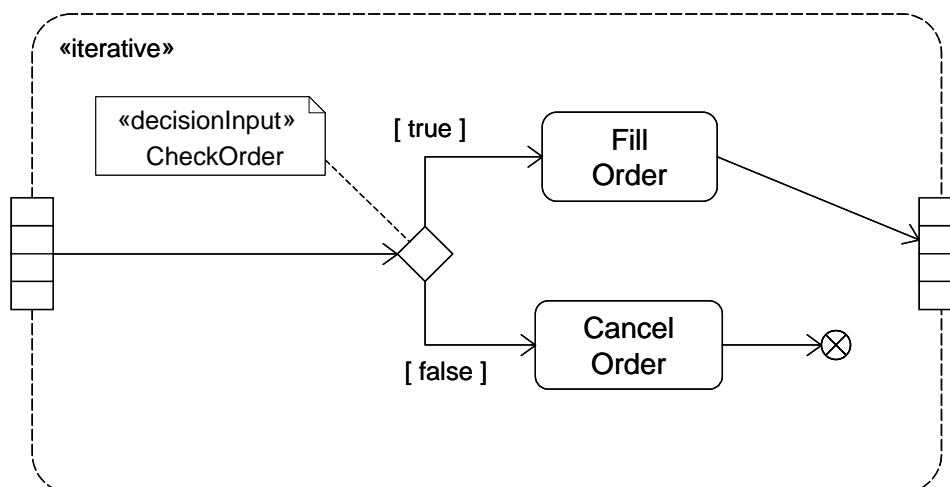


Figure 17: UML Notation for Expansion Regions

```
(dolist (o orders filled-orders)
  (cond ((CheckOrder o)
        (setq filled-orders
              (nconc filled-orders '(o))))
        (t (CancelOrder o))))
```

Figure 18: Example Textual Notation for Figure 17

<sup>15</sup> Expansion regions replace the UML 1.5 filter action, map action, and iterate action. They do not replace UML 1.5 reduce action, which transforms a collection into a scalar by repeated binary combination of the input elements. It was inadvertently dropped in UML 2.0 and will be returned in revision.

The expansion region in is marked as being in ITERATIVE mode, so each order is filled or cancelled before the previous one is checked. Alternatively, it could have processed orders in parallel, or as a stream. In PARALLEL mode, the elements of the collection move through “copies” of the region and do not interact with each other. If the example used this mode, orders that are quickly filled and checked could be processed in parallel with orders that take longer. In STREAM mode, elements enter the same “copy” of the region one after the other. If the example used this mode, it would allow checking an order in parallel with filling the order that came before it, informally called “pipelining.”<sup>16</sup> In all modes, the entire region completes when all the elements of the input collection have been acted on by the region. Figure 19 shows most of the repository model for and Figure 18, omitting parameters and variables for brevity, as well as the containment links from the expansion node the elements in it. Expansion nodes are specified with the inputElement and outputElement associations.<sup>17</sup> They can be the source and target of object flows, as all object nodes can.

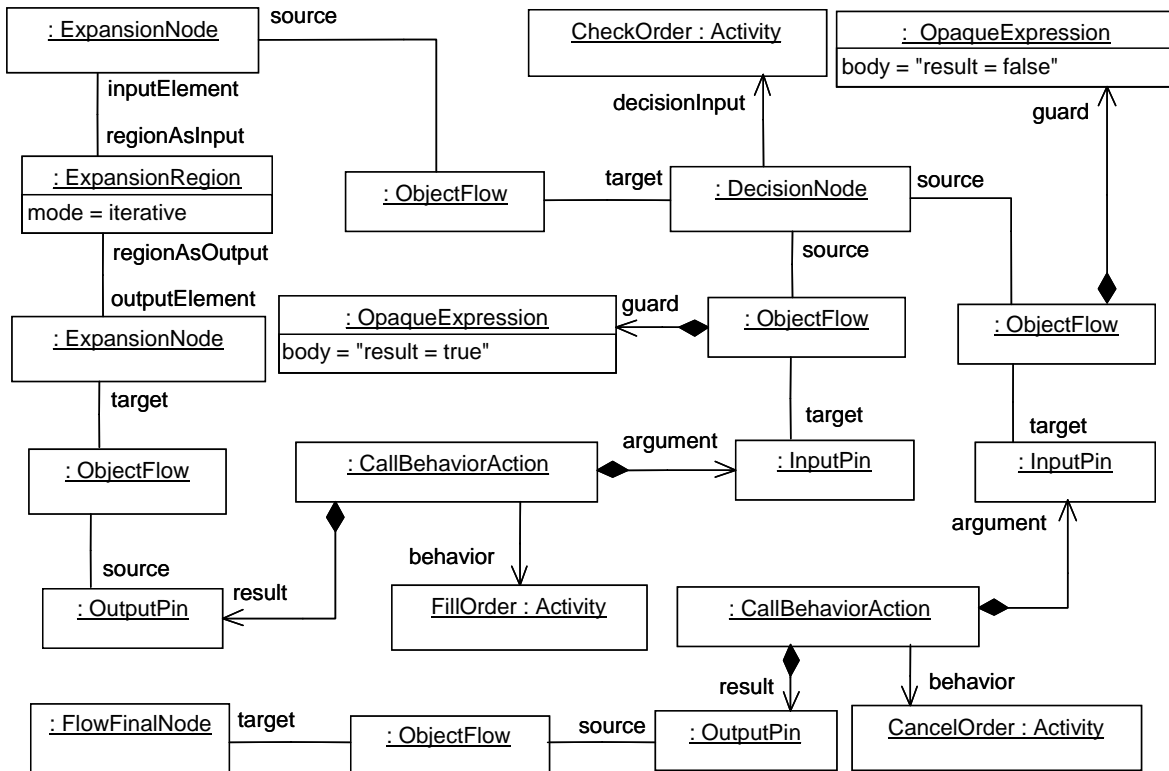


Figure 19: Repository Model for Figure 17 and Figure 18

It is common to have a region that simply calls a behavior or operation on each element

<sup>16</sup> The activity model supports hybrid stream/parallel modes where values upstream can overtake other values downstream. This means some of the actions in the activity allow concurrent executions, which is indicated by the ISREENTRANT property on invoked behaviors. This will be described later in the series.

<sup>17</sup> The INPUTELEMENT and OUTPUTELEMENT associations are not specialized from the INPUT and OUTPUT associations for pins, because the semantics of expansion nodes are not the same as pins, as described above and in the bullet above Figure 22.

of the collection and outputs a collection of the return results. Two standard UML shorthand notations are available for this, as shown in Figure 20. These examples call a single behavior for processing orders, which returns the order modified to indicate whether it was successfully processed or not. They are equivalent to the full notation in Figure 21. The shorthand at the top of Figure 20 supports all modes, while the one at the bottom always translates to parallel mode.<sup>18</sup>

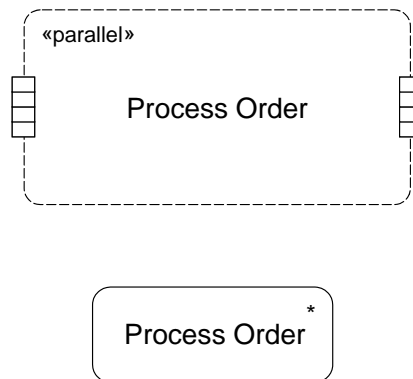


Figure 20: Shorthand UML Notations for Expansion Regions

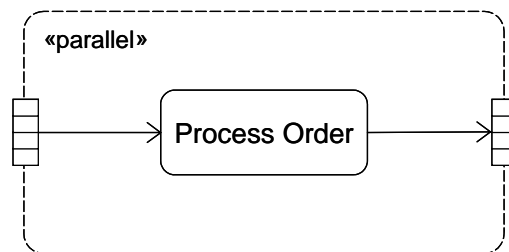


Figure 21: Longhand UML Notation for Figure 20

Expansion regions have more general features than illustrated so far:

- Input collections can be ordered. In iterative mode, the region will act on the elements according to the order specified by the collection. The stream mode will feed elements into the region in that order also.
- A region can accept multiple collections and output multiple collections. Each execution of the region draws an element from each input collection into the same “copy” of the region, but may act on elements across collections according to mode. For example, Figure 22 shows multiple collections for a fragment of the Fast Fourier transform algorithm, adapted from Figure 261 of [1] and Figure C-24 of [5]. An execution of the region takes one element from each input collection, LOWER, UPPER, and ROOT, which is informally called a “slice.”<sup>19</sup> Since the region

<sup>18</sup> In UML 1.5 the asterisk in the lower notation of Figure 20 specified a multiplicity constraining the number of concurrent executions of the behavior. In UML 2 it is purely notational.

<sup>19</sup> UML does not currently specify whether the name and type of expansion nodes refer to collections, as the values appear outside the region, or to the elements, as they appear inside. This will be addressed in revision. Figure 22 uses the name and type of the elements.

is in parallel mode, it can act on all slices through the inputs simultaneously. Multiple input collections must have the same number of elements at runtime when execution starts on an expansion region. The number of input collections may differ from the number of output collections, as in Figure 22, and the type of elements in each collection do not need to be the same.

- Values flowing into the region without going through an expansion node are taken as constant inputs to executions of the region. The same applies to values arriving on input pins (expansion nodes are not pins). This is a weak form of loop variable, where the values cannot be modified.

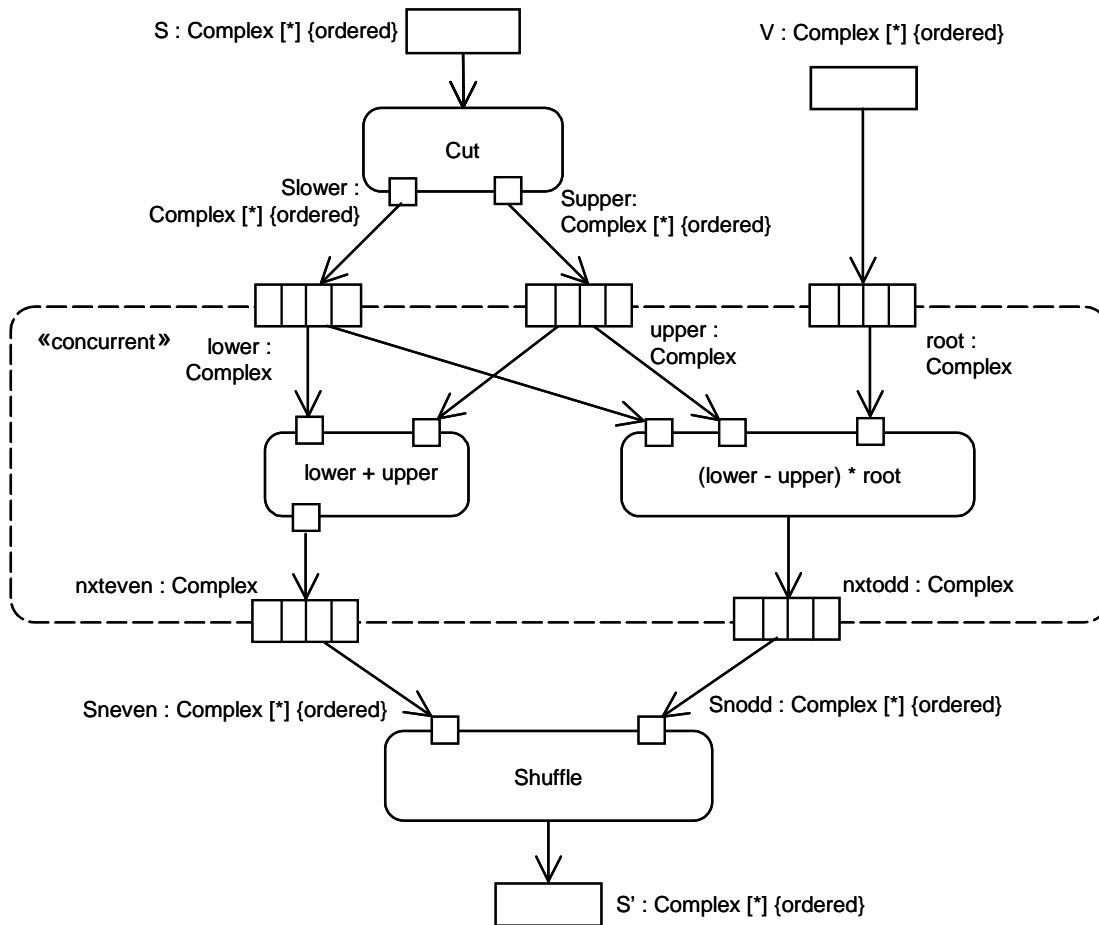


Figure 22: Expansion region with Multiple Input Collections



## 7 EXCEPTION HANDLING

UML 2 has two exception handling facilities, one typically used with structured models and the other for flow models. The structured one is analogous to try/throw/catch constructs in programming languages. It provides a way to indicate that a structured node or action traps exceptions raised from inside it or from behaviors it calls. An exception in UML is any object thrown with the predefined action `RAISEEXCEPTIONACTION`. An example in standard UML notation is shown in Figure 23, omitting parameters and the contents of the structured node for brevity, with an example textual notation from C++ [11] in Figure 24, and repository model in Figure 25. It assumes the `CHECKORDER` behavior will raise an exception of type `NOFILLREASON` if the order does not pass the check. When this happens, all tokens flowing in the execution of `CHECKORDER` and the node invoking it are destroyed. The zigzag arrow to the input pin of `NOTIFYBUYER` indicates the structured node traps exceptions of type `NOFILLREASON`, and the reaction will be to notify the buyer that something is wrong with the order.<sup>20</sup>

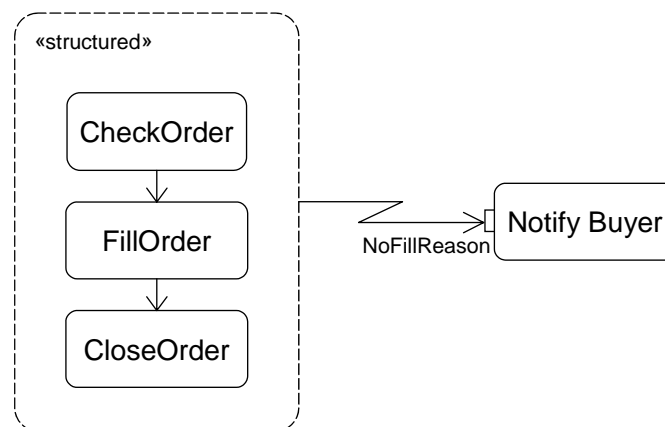


Figure 23: UML Notation for Exception Handlers

```
try
{
    CheckOrder();
    FillOrder();
    CloseOrder();
}
catch (NoFillReason *r)
    NotifyBuyer(r);
```

Figure 24: Example Textual Notation for Exception Handlers

<sup>20</sup> An alternative notation for the zigzag line is to use a zigzag icon above a straight line. See Figure 254 of [1].

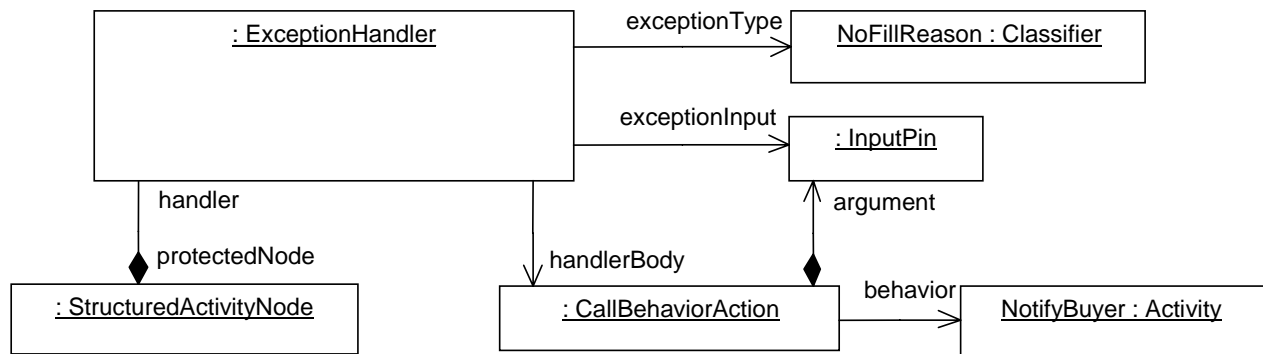


Figure 25: Repository Model for Figure 23 and Figure 24

In general, an exception is passed from the point at which it is raised, up the “call tree” through all containing structured nodes, activities, synchronous call behavior and operation actions, until it reaches a structured node or action protected by an exception handler for the type of exception raised. All tokens are destroyed in constructs the exception object passes through on the way up to the handler. Then the handler is run. UML does not specify what happens if no handler is found for an exception at all.<sup>21</sup> Exceptions are not passed up through asynchronous invocation actions. These actions do not expect a reply, and separate the caller and callee completely. If the protected node has an output pin, and an exception is thrown, the handler output value is used. This makes exception handlers more general than typical programming constructs, because they can be used on expressions as well as statements.<sup>22</sup>

Two exception handling facilities for flow models provide for the abandonment of an activity or portion of it when some values pass outside it. The first is shown in Figure 26, where a portion of an activity is indicated as an interruptible region with an interrupting edge shown as a zigzag line.<sup>23</sup> When a value flows along the interrupting edge, all tokens in the region are destroyed.<sup>24</sup> In this example, the interrupting value happens to be a signal of type NOFILLREASON sent by the lower level activities when there is a problem filling the order, and received by the overall activity.<sup>25</sup> In general, the signal can be sent from any activity, not just the ones called in the region, as with exceptions thrown to handlers. The repository model for interruptible regions is shown in Figure 25, omitting the order actions for brevity. It is similar to those of structured nodes, except that it

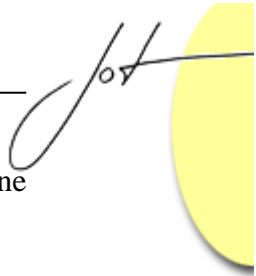
<sup>21</sup> The current specification is not clear about the effect of multiple handlers matching the exception, but it is expected to be clarified as choosing one of the handlers indeterminately.

<sup>22</sup> Exception handlers replace UML 1.5 jump handlers. UML 1.5 used exceptions to model programming constructs for non-local control flow, such as breaks and continues. This is not necessary, since they have the same effect as an unstructured control flow.

<sup>23</sup> An alternative notation for an interrupting edge is to place a zigzag icon above a straight line. See Figure 273 of [1].

<sup>24</sup> The term “interruptible” region is a misnomer, because the region cannot be restarted with the same token state as when it was terminated.

<sup>25</sup> The signal may be sent to the object executing the overall activity or to the activity itself, which is also an object [2].



identifies the edges that can interrupt it, and it allows a node to be in more than one interruptible region.

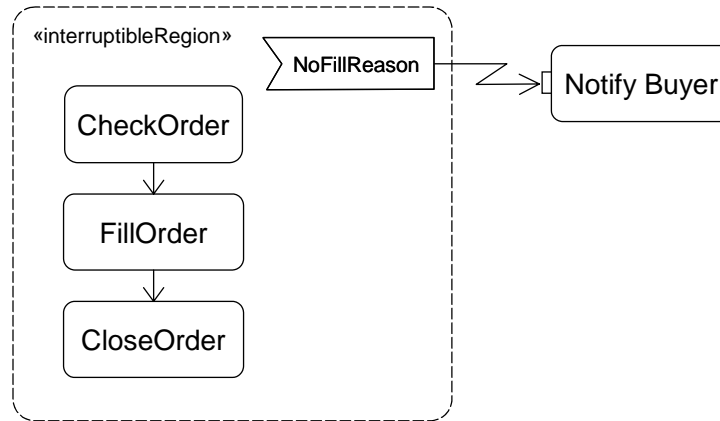


Figure 26: UML Notation for Interruptible Regions

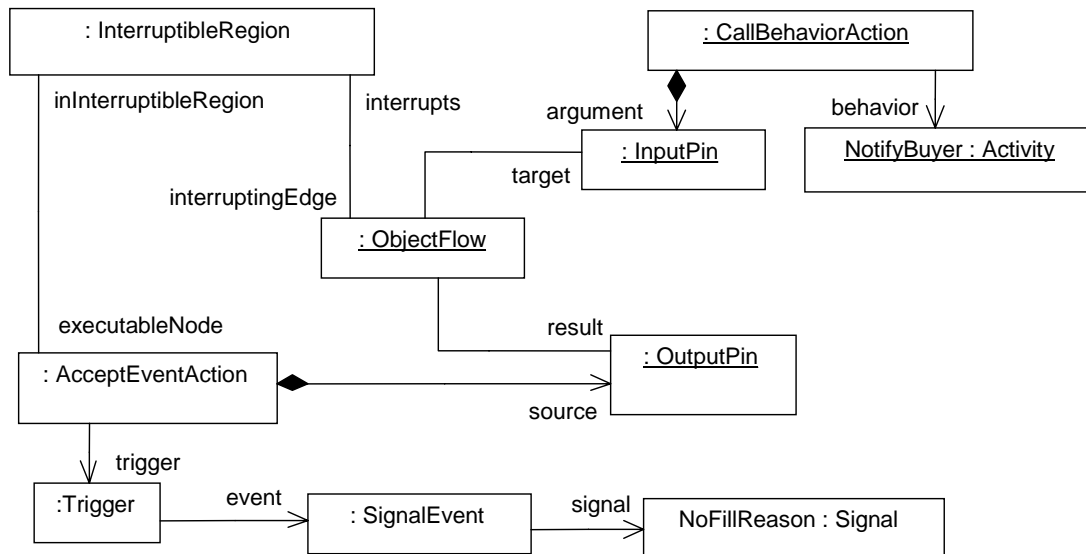


Figure 27: Repository Model for Figure 26

The second exception handling facility for flow models is exception parameters. These provide output values to the exclusion of any other output parameter or outgoing control of the action. They destroy all tokens in the activity or action they flow out of. Figure 28 shows an activity for processing orders, using the triangle annotation for an exception parameter. If a NOFILLREASON signal arrives while the activity is executing, it flows to the exception parameter, which terminates the activity. The order is not output. If the signal does not arrive, the order is output and the exception is not. These are also described in an earlier article of the series, see Figure 10 of [6].

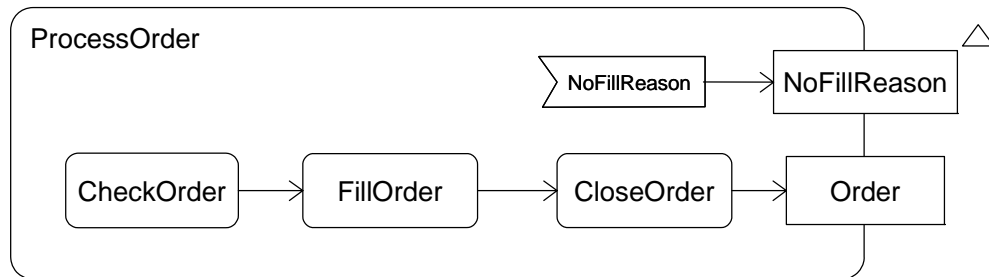


Figure 28: Exception Output Parameter

## 8 EXPRESSION TREES

Most textual languages provide for nested expressions to pass results of one function to another without using variables. For example, Figure 29 shows an expression used to calculate the value of a variable. This is a natural application of data flow models, since there are no variables for intermediate results in the expression, like the sum of X and 1. Figure 30 shows the equivalent UML notation.<sup>26</sup> The textual and graphical notations can be compiled to the same underlying repository model.

$$y = x / (x + 1)$$

Figure 29: Example Textual Notation for Expression Trees

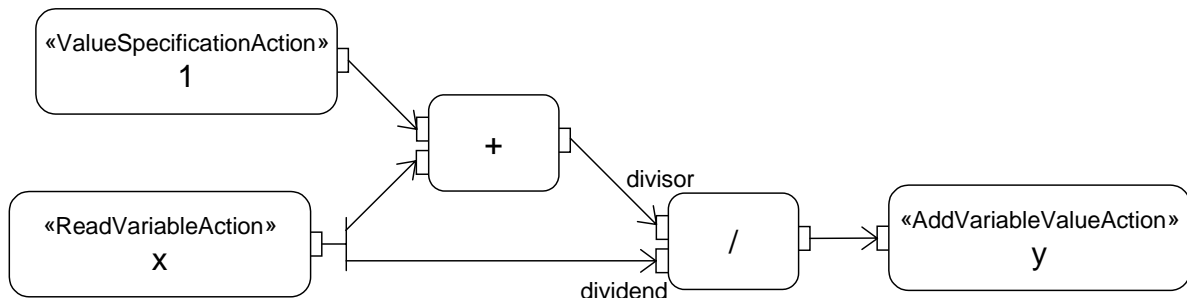
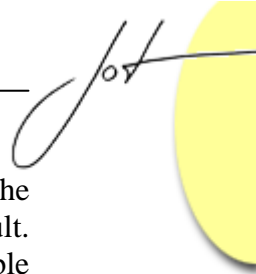


Figure 30: Equivalent UML Notation for Figure 29

<sup>26</sup> UML provides for more compact notation for reading variables and injecting value specifications, by placing the variable and value specification near the input pins that use their values.



When nested expressions are translated to activities, control first arrives at leaves of the expression tree, after which data cascades through to the root, producing the final result. For example, in Figure 30, control is passed to the value specification and read variable actions on the left, and the final result is produced from the division action. This contrasts with the expectation of some compiler authors, who expect control to start at the statement that requires the expression, and the expression to be evaluated from the root down. In this view, control would arrive at the add variable action in Figure 30, which would begin the evaluation of the division expression, which would begin the reading of the X variable, and so on to the leaves of the tree. This is a “pull” style of data flow, where an action requiring data initiates other actions that provide it, as compared to the “push” style of Figure 30, where actions that produce data initiate other actions that accept it.

For those preferring pull data flow for expression trees, UML 2 provides a specialized form of input pin that invokes an action when the pin needs a value. These action input pins are only invoked when all the other inputs to the action are already available. The action being invoked must have exactly one output pin. Since action input pins are for parsing textual languages, they do not have a standard UML notation. The repository for an action pin model of Figure 29 is shown in Figure 31. When control arrives at the action for setting the Y variable, an action input pin for the value initiates the division action, which has action input pins initiating other actions, and so on. The leaves of the expression tree only have actions that have no inputs. Action input pins are a generalization of value pins, introduced in Figure 4. A value pin is equivalent to using a value specification action with an action input pin, and has the same pull semantics.

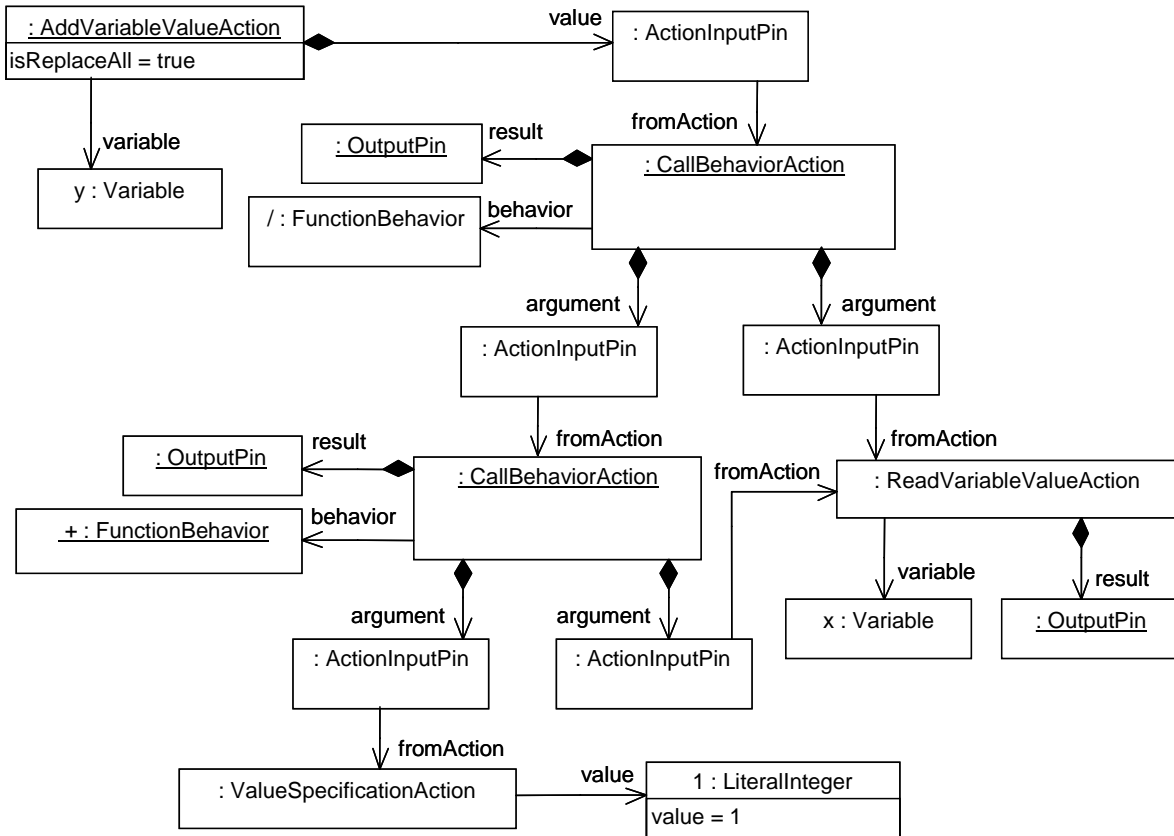


Figure 31: Repository Model for Figure 29 Using Action Pins

## 9 CONCLUSION

This is the sixth in a series on the UML 2 activity and action models. It covers models for languages that usually have textual presentations, including structured nodes for sequencing, conditionals, loops, and expansion regions for operating on collections, as well as exception handlers, variables, and action pins. Examples are given in various nonstandard textual formats, because UML does not specify a textual notation, along with a repository model to show how they translate to the activity model. Expansion regions provide for mapping and filtering collections in multiple modes, including concurrency. Exception handling and expression trees are supported in three ways, one for structured models and another two for flow models. It is explained how each construct is more general than the corresponding ones in typical programming languages.



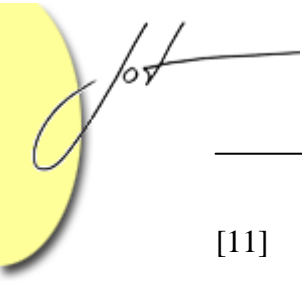
## ACKNOWLEDGEMENTS

Thanks to James Rumbaugh for input to this article and the original merger of UML 1.5 actions into UML 2 activities, and to Evan Wallace and James Odell for their reviews.

Commercial equipment and materials might be identified to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

## REFERENCES

- [1] Object Management Group, "UML 2.0 Superstructure Specification," October 2004. <http://www.omg.org/cgi-bin/doc?ptc/04-10-02>
- [2] Bock, C., "UML 2 Activity and Action Models," in *Journal of Object Technology*, vol. 2, no. 4, July - August 2003, pp. 43-53, [http://www.jot.fm/issues/issue\\_2003\\_07/column3](http://www.jot.fm/issues/issue_2003_07/column3)
- [3] Sharman, D. and Yassine, A. "Characterizing complex product architectures," *Journal of the International Council on Systems Engineering*, vol. 7, no. 1, pp.35-60, February 2004.
- [4] Bock, C., "UML Without Pictures," *IEEE Software Special Issue on Model-Driven Development*, vol. 20, no. 5, pp. 33-35, September/October 2003.
- [5] Object Management Group, "OMG Unified Modeling Language," version 1.5, March 2003. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>
- [6] Bock, C., "UML 2 Activity and Action Models, Part 2: Actions," in *Journal of Object Technology*, vol. 2, no. 5, pp. 41-56, September-October 2003. [http://www.jot.fm/issues/issue\\_2003\\_09/column4](http://www.jot.fm/issues/issue_2003_09/column4)
- [7] Bock, C., "UML 2 Activity and Action Models, Part 3: Control Nodes," *Journal of Object Technology*, vol. 2, no. 6, pp. 7-23, November - December 2003. [http://www.jot.fm/issues/issue\\_2003\\_11/column1](http://www.jot.fm/issues/issue_2003_11/column1)
- [8] Kernighan, B., Ritchie, D., *The C Programming Language*, Second Edition, Prentice Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1988.
- [9] Graham, P., *ANSI Common LISP*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1995.
- [10] Bock, C., "UML 2 Activity and Action Models, Part 4: Object Nodes," *Journal of Object Technology*, vol. 3, no. 1, pp. 27-41, January - February 2004. [http://www.jot.fm/issues/issue\\_2004\\_01/column3](http://www.jot.fm/issues/issue_2004_01/column3)



- [11] Kalev, D., ANSI/ISO C++ Professional Programmer's Handbook, Que, 1999.

### About the author



**Conrad Bock** is a Computer Scientist at the U.S. National Institute of Standards and Technology, specializing in process models and ontologies. He is one of the authors of UML 2 activities and actions, and can be reached at [conrad.bock@nist.gov](mailto:conrad.bock@nist.gov).