JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering **©JOT, 2003**

Vol. 2, No. 4, July-August 2003

UML 2 Activity and Action Models

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

This is the first 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 series is a companion to the standard, providing additional background, rationale, examples, and introducing concepts in a logical order. This article covers motivation and architecture for the new models, basic aspects of UML 2 activities and actions, and introduces the general notion of behavior in UML 2.

1 BACKGROUND

A focus of recent developments in UML has been on procedures and processes. UML 1.5 introduced a model for parameterized procedures defined by control and data flow to complement the existing state and interaction models [2]. For the first time UML supports first-class procedures that can be used as methods on objects or independently of objects, as occurs when modeling, for example, function libraries with popular programming languages. It also facilitates application of UML by modelers who do not use object-orientation (OO) routinely, such as system engineers and enterprise modelers, and provides them a path to incrementally adopt OO as needed [3]. The flexibility to combine OO with functional approaches considerably widens and integrates the potential applications of UML.

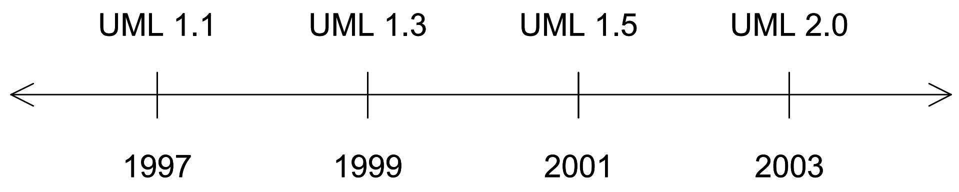


Figure 1: UML Timeline

UML 1.5 also inherited from earlier versions a form of state machine that was notationally modified to appear as a flow diagram, called the activity diagram. Unfortunately, the underlying state machine semantics restricts expressiveness and is confusing to users. Especially those not using an OO approach perceive UML 1.x activity models as not working for them. There is also concern that UML 1.5 has multiple models for control and data flow.

In light of this experience, UML 2.0 redefined the activity model to give it a basis in flow modeling intuition, and to integrate it with the UML 1.5 action model. The new activity model supports the control and data flow of UML 1.5 procedures, as well as more general features, such as cycles and queuing. Consequently, UML 2 activities support flow modeling across a wide variety of domains, from computational to physical. This makes it ideal for specifying systems independently of whether the implementation is software or hardware, as in system engineering, and independently of where the system/environment boundary is drawn. Finally, the combination of activities and actions retains the UML 1.x capability of reacting to events, so can be applied to areas requiring that, such as embedded and agent-based systems.

2 MULTIPLE DOMAINS

Although the UML 2 activity and action models are defined independently of application, some features are more appropriate to some domain styles than others. For example, actions for throwing exceptions will probably be used more often by programmers, whereas enterprise modelers are more likely to apply other techniques for atypical flows, such as exception parameters and interrupting regions. This redundancy is unavoidable when creating an abstraction over user groups that do not overlap. However, it is more than made up for by the efficiency in communication based on a common model between domains that are integrated in delivered systems.

To support vendors focusing on particular users, the activity model is packaged in a more fine-grained way than other behavior models, and has more complex dependencies between packages, as shown in Figure 2.

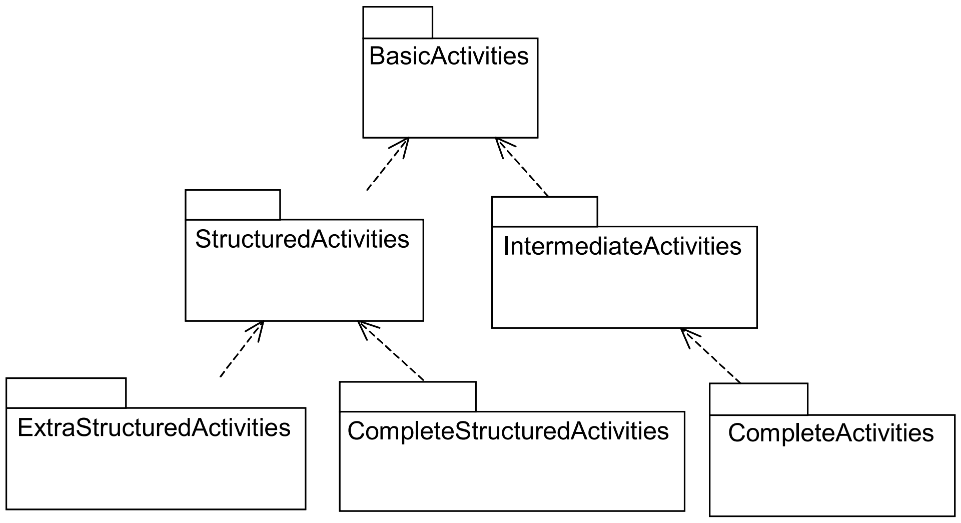


Figure 2: Activity Package Dependencies

The features in common between the applications, such as the abstract action concept, control/data flow, and input/output, are at the root (basic activities). From there, dependencies branches in two directions, one mainly for software modeling (structured activities), and the other for general process modeling (intermediate and complete activities). Structured activities introduce well-nested constructs, for modeling conditionals, variables, try/catch, and so on. Intermediate and complete activities introduce explicit parallelism, partitions, flow-based forms of exception handling, and a number of constructs for finer-grained control of flows. These are orthogonal branches, so they may be combined. For example, structured activities may be folded in with intermediate activities, to support explicit parallelism and structured conditionals at the same time.[1] This series will cover the functionality of the various packages.

Supporting a wide range of applications also requires multiple notations. For example, programmers tend to favor textual notations, while subject matter experts prefer graphical notations. UML addresses this by defining a repository for storing specifications that can be populated from multiple notations. Programmers can use a textual syntax to build and read activity models in the repository, and enterprise modelers can use a graphical notation. The UML repository is a communication medium between multiple notations, and a source for highly directable compilers targeting multiple platforms [4].

3 FLOW MODELS AND SEMANTICS

Behavior models in general determine when other behaviors should start and what their inputs are [3]. In particular, the UML 2 activity models follow traditional control and data flow approaches by initiating subbehaviors according to when others finish and when inputs are available. It is typical for users of control and data flow to visualize runtime effect by following lines in a diagram from earlier to later end points, and to imagine control and data moving along the lines. Consequently a token flow semantics inspired by Petri nets is most intuitive for these users, where "token" is just a general term for control and data values.

UML 2 takes the same approach to behavioral semantics as UML 1.x, namely to define intuitive virtual machines. This enables users and vendors to predict the runtime effect of their models. UML 2 activities define a virtual machine based on routing of control and data through a graph of nodes connected by edges. Each node and edge defines when control and data values move through it. These token movement rules can be combined to predict the behavior of the entire graph.[2] The rules for control and data movement are only intended to predict runtime effect, that is, when behaviors will start

[1] The action model should be finely packaged to correspond with activities, but is currently divided into INTERMEDIATEACTIONS and COMPLETEACTIONS only. These contain the predefined actions, such as object creation, setting attributes, and so on. The abstract notion of action is defined in the BASICACTIVITIES package because it is integral to the flow model. See later sections of this article.

[2] It is hoped that the rules are precise enough to be translated to a formal semantics, especially to support proving properties about modeled processes. This is left for future work.

and with what inputs. They do not constrain implementation further than that. In particular, the rules do not imply that activity models must be implemented as a virtual machine corresponding to the token analogy, messaging passing, or any other scheme. It is only necessary that the runtime effects predicted by the virtual machine actually occur in the implementation.

The activity model is defined with few semantic variations, which is the UML term for allowing implementations to choose alternative runtime behavior without recording those choices in the model. Semantic variations cause a model in one implementation to execute differently than the same model in another implementation, with no standard way to tell what the differences are. Variations in activity execution are mostly modeled as attribute values in the user's model. This means they are under user control and are transmitted to other users without requiring implicit alignment of implementations.

4 ACTIVITY NODES AND EDGES

UML 2 activities contain nodes connected by edges to form a complete flow graph. Control and data values flow along the edges and are operated on by the nodes, routed to other nodes, or stored temporarily. More specifically, there are three kinds of node in activity models:

1. Action nodes operate on control and data values that they receive, and provide control and data to other actions.
2. Control nodes route control and data tokens through the graph. These include constructs for choosing between alternative flows (decision points), for proceeding along multiple flows in parallel (forks), and so on.
3. Object nodes hold data tokens temporarily as they wait to move through the graph.

Figure 3 shows the notation for some of the activity nodes to be discussed. Contrary to the names, control nodes coordinate both data flow and control flow in the graph, and object nodes can hold both objects and data.[3]

Activity nodes are connected by two kinds of directed edges:

1. Control flow edges connect actions to indicate that the action at the target end of the edge (the arrowhead) cannot start until the source action finishes. Only control tokens can pass along control flow edges.
2. Object flow edges connect objects nodes to provide inputs to actions. Only objects and data tokens can pass along object flow edges.

[3] UML abstracts from objects and data to classifiers in general.

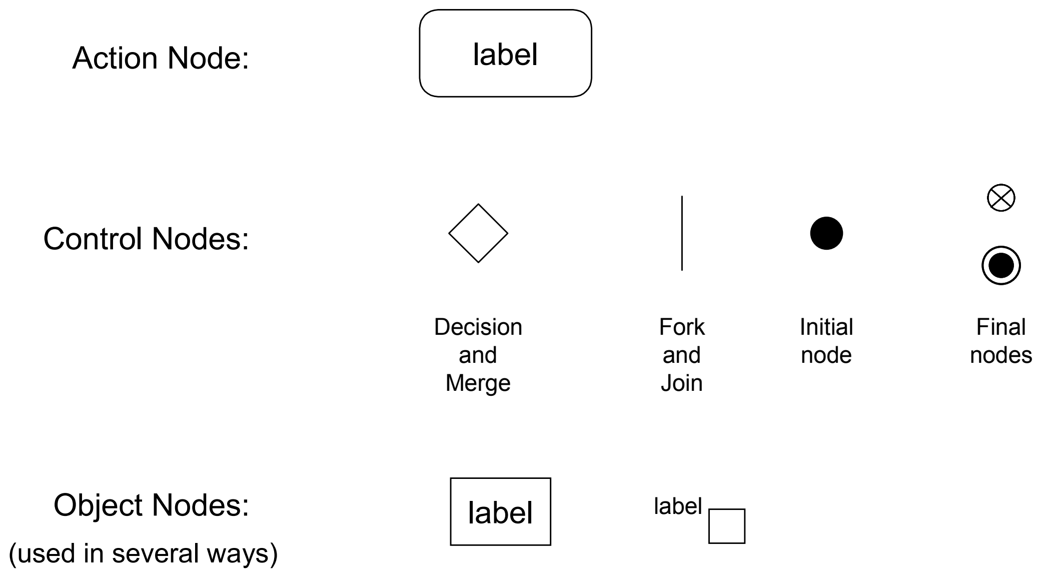


Figure 3: Activity Nodes

Figure 4 shows the notation for edges. Control and object flow edges are distinguished by usage. Control edges connect actions directly, whereas object flow edges connect the inputs and outputs of actions (see next section).

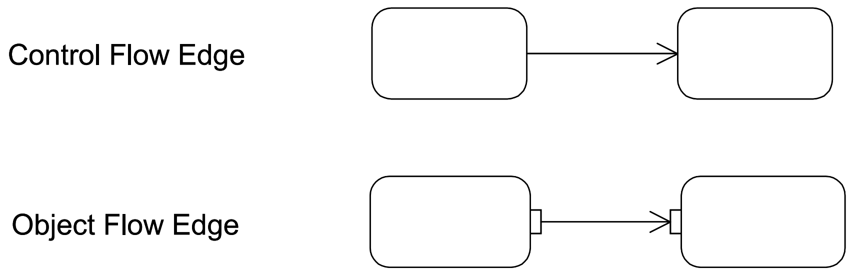


Figure 4: Activity Edges

This rest of this article introduces actions and edges with a small example. Later articles will cover the other kinds of nodes, and more features of actions and edges.

5 ACTIONS

Activity models coordinate actions, some of which may invoke user-defined behaviors, including other activities. All actions are predefined. For example, UML 2 has actions to create objects, set attributes values, link objects together, and to invoke user-defined behaviors. Actions can have inputs and outputs, which are called pins, that are connected by object flow edges to show how values flow through the activity, provided by some actions and received by others. In the simple cases, all inputs to an action are required to be available for it to begin executing.

Figure 5 shows an example of an action creating a new instance of an ORDER class, then another action invoking a user-defined behavior to fill it. The object creation action creates a blank order that FILLORDER completes. Action nodes are notated with round cornered rectangles.[4, 5] At the top of Figure 5, the small rectangles attached to the actions are input and output pins. The type of object accepted as input or provided as output is shown as an adornment. The notation at the bottom of Figure 5 can be used when the types of input and output are the same. Pins are a kind of object node, so they also hold data values temporarily in the flow.

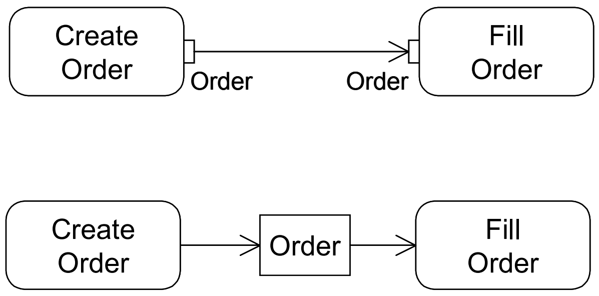


Figure 5: Example Actions and Object Flow Edges

As mentioned before, it is not necessary to use the notation of Figure 5. Programmers will mostly likely prefer a textual notation [4], such as:

```
Order o;
o = new Order;
FillOrder(o);
```

The repository model defined by UML 2 is the same for the above notations, as shown in Figure 6, assuming the variables of the textual language are modeled as dataflow.[6] Each element of the repository is an instance of a metaclass defined in the UML specification, which is the name to the right of the colon. The name of the repository instance itself is to the left of the colon, blank if it is anonymous. The model shows a CREATEOBJECTACTION with an output pin passing its value to the input pin of a CALLBEHAVIORACTION.. The

[4] The wording inside the action nodes is not normative, because a standard textual notation for actions is not adopted yet. Also action nodes can be given labels that are more descriptive than the predefined action name.

[5] The round cornered notation is also used by state machines to notate states, unfortunately. This is not a desirable situation. However, it is beneficial to those users and vendors who have been trying to apply state machines to flow modeling applications, for lack of a flow modeling standard until now.

[6] UML 1.5 and UML 2 also support variables if needed.

CREATEOBJECTACTION is linked to the user class it instantiates (ORDER), and the CALLBEHAVIORACTION is linked to the user-defined behavior it invokes (FILLORDER).

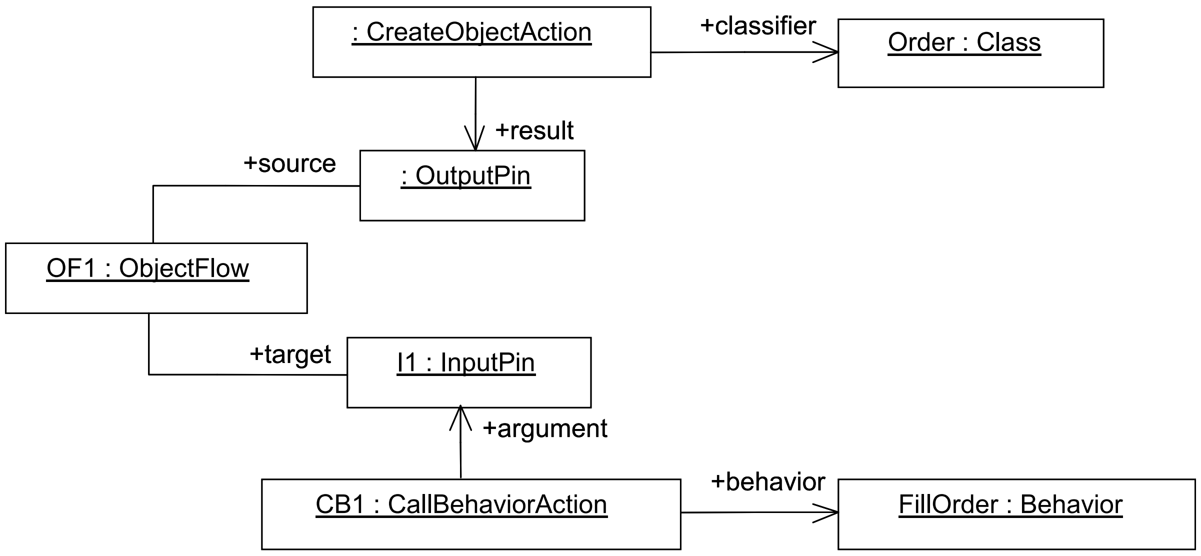


Figure 6: Repository model for Figure 5

The primitive actions for creating objects, invoking user-defined behaviors, and so on, are not technically behaviors themselves, but this is more an artifact of metamodeling than a conceptual distinction. The fundamental distinction is between a specification of dynamic effect and its usage, to support multiple usages of the same specification. For example, the same FILLORDER behavior may be invoked in many activity diagrams, or many times in the same activity diagram, but each invocation will be represented by a separate instance of CALLBEHAVIORACTION in the repository, all referring to the same FILLORDER behavior. This is because each usage of FILLORDER will be in a different flow, or at different points in the same flow, and each usage may have different actions before and after it. For example, Figure 6 has a CREATEOBJECTACTION before FILLORDER, whereas another flow might not.[7]

Actions are also reusable, but this happens to be modeled in a different way than user-defined behaviors. Each action in a flow is a new instance of a single class from the UML metamodel. For example, if an activity contains two object creation action nodes, then the user's repository has two instances of the CREATEOBJECTACTION class from the UML metamodel, and separate sets of pins for each. Reusability is achieved by using multiple

[7] Programming languages make the same distinction between a procedure declaration or definition, which has the signature, and statements that call the procedure, providing actual parameters at runtime. In UML 2 terms, a procedure definition is a behavior, and a statement is an action.

instances of the same UML metaclass as action usages[8]. Later articles will cover the various kinds of actions.

6 ACTIVITIES

Activities are user-defined behaviors, and like all behaviors in UML 2 can be initiated by invocation actions, and support parameters to receive and provide data to the invoker. Parameters are part of the reusable definition of an activity. Parameters are not pins, because pins are used to connect actions in a flow, whereas activities are behaviors invoked by actions (see previous section). However, to access parameter values from actions in the activity, activity parameters are modeled as a special kind of object node used to temporarily hold actual parameter values as they flow into and out of the activity. Figure 7 shows a parameterized activity with a parameter object node connected to pins on actions. Parameter object nodes are shown on the border, with object flow edges connecting them to pins. The type of object held in an object node is usually shown in the label. In this example, the information used to fill the order is provided as an input parameter and passed along to the invocation of the FILLORDER behavior.

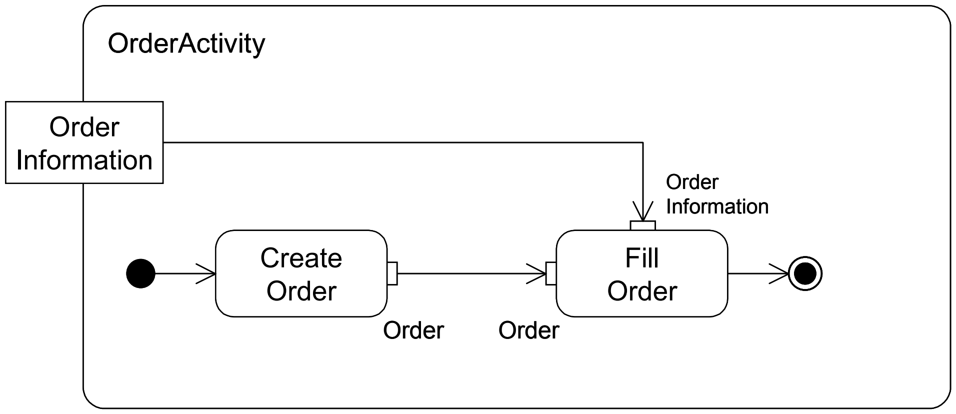


Figure 7: Example Activity

The control nodes at the beginning and end of the flow in Figure 7 are initial and final nodes respectively. When the ORDERACTIVITY is invoked, a control token is placed at the initial node, and a data token with ordering information is placed at the input parameter object node. The control token flows from the initial node to the CREATEORDER action, which begins executing. The data token flows from the parameter to the invocation action

[8] It is possible that actions could be modeled in the same way as user-defined behaviors and be standardized as a reusable model library. Then only one predefined action would be needed to invoke all behaviors, whether predefined or user-defined. However, the static requirements of predefined actions, such as CREATEOBJECTACTION requiring a class to instantiate, would need to be recorded in constraints rather than associations in the metamodel. Constraints are generally not as easy to read in the UML specification as metamodel associations.

for FILLORDER, which must wait for CREATEORDER to provide its other input before starting. When FILLORDER is done, a control token is passed to the final node and the activity terminates, returning control to execution that started it. A partial repository model for Figure 7 is in Figure 8 below. It connects to the repository model of Figure 6 through the CB1 CALLBEHAVIORACTION. The CREATEORDER action and its flows are omitted for brevity.

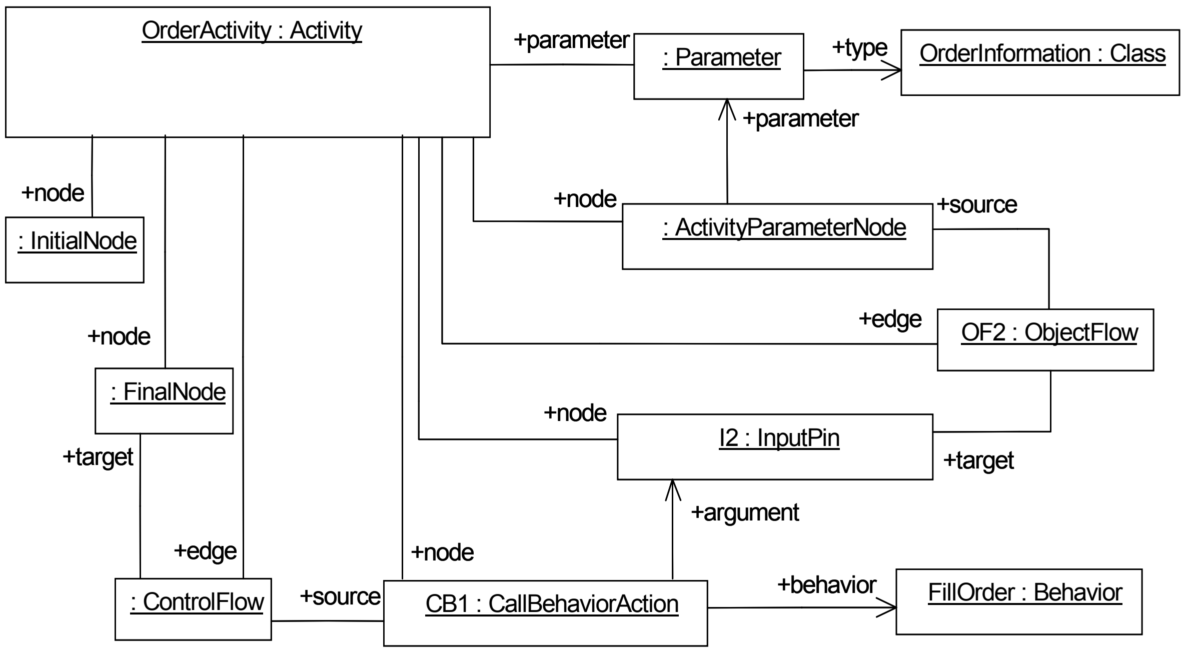


Figure 8: Partial repository model for Figure 7

7 BEHAVIOR IN UML 2

UML 2 supports the concept of parameterized behavior for all the kinds of behavior in UML, not just activities. This means state machines, interactions, and activities all can be parameterized, and be methods on objects or invoked directly, in a uniform way. The upper right of Figure 9 shows an activity model for a behavior called DELIVERMAIL (the curved arrows are not part of UML notation). DELIVERMAIL could be invoked as is with a CALLBEHAVIORACTION, or as a method on the POEMPLOYEE class with a CALLOPERATIONACTION. In either case, the behavior takes an instance of KEY as input. Because behaviors can be invoked directly or as operations, UML 2 provides a path to incrementally adopt object-orientation [3].

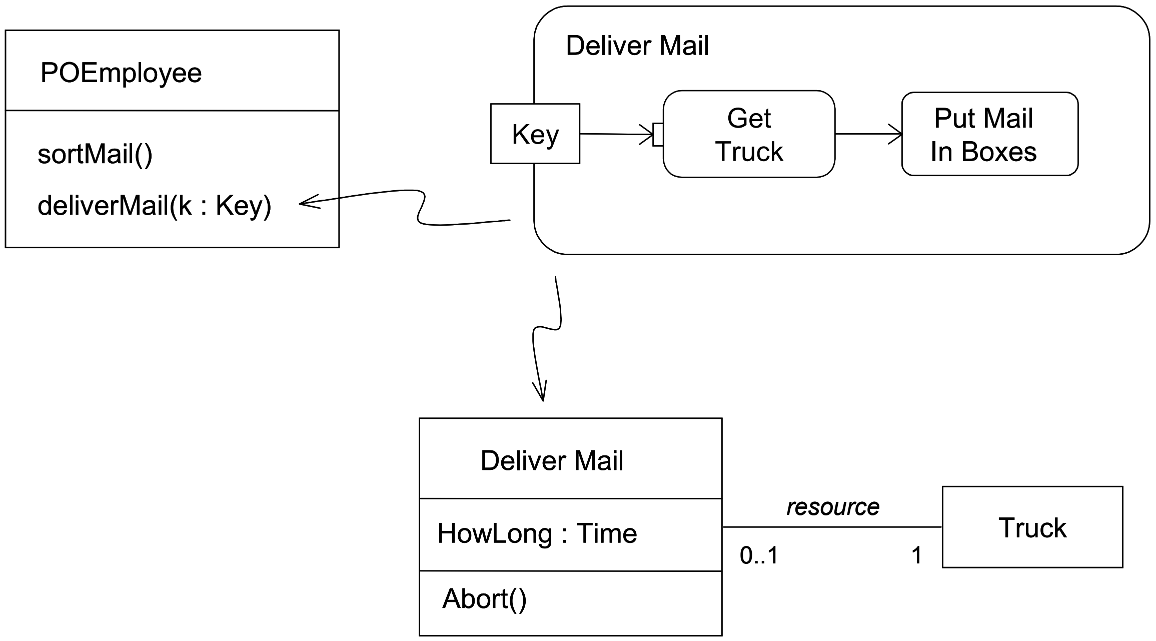


Figure 9: UML 2 Behavior

UML 2 user-defined behaviors are also classes. Each time a behavior is executed at runtime, a new runtime instance of the user's behavior class is created. The instance is destroyed when the behavior terminates. Behavior classes, like all classes, can support attributes, associations, operations, and even other behaviors, such as state machines. This reflects common practice in systems that manage processes, for example, workflow and operating systems. The bottom of Figure 8 shows the behavior class for the DELIVER MAIL activity with an attribute for how long each execution of DELIVER MAIL has been running, an operation to abort the execution, and an association for the truck it is using. Applications can also put state machines on the behavior class to describe the status of each execution, such as NOT_STARTED, SUSPENDED, and so on [5] [6].

UML 2 behavior classes enable the definition of standard functionality for process management, even though UML 2 does not define standard features itself. Behavior class features can be defined by domain standards, vendors, or user groups as reusable model libraries containing abstract behavior classes with normative attributes and operations such as ABORT and so on. Then these classes can be used as supertypes of user-defined behaviors such as DELIVER MAIL in Figure 9.

8 CONCLUSION

This article begins a series on the UML 2 activity and action models. It reviews progress in UML flow modeling, package structure needed to serve the wide range of flow modeling applications, UML's approach to semantics, then introduces some basic elements of activity modeling, and the UML 2 behavior model generally.

ACKNOWLEDGEMENTS

Thanks to Evan Wallace and James Odell for their input to this article.

REFERENCES

[1] U2 Partners, "Unified Modeling Language: Superstructure", version 2.0, 3rd revised submission to OMG RFP ad/00-09-02, http://www.omg.org/cgi-bin/doc?ad/2003-04-01, April 2003.

[2] Object Management Group, "OMG Unified Modeling Language", version 1.5, http://www.omg.org/cgi-bin/doc?formal/03-03-01, March 2003.

[3] Bock, Conrad, "Three Kinds of Behavior Model," *Journal of Object-Oriented Programming*, 12:4, July/August 1999.

[4] Bock, Conrad, "UML Without Pictures", to appear in *IEEE Computer Special Issue on Model-driven Development*, September/October 2003.

[5] Object Management Group, "Workflow Management Facility Specification", version 1.2, http://www.omg.org/cgi-bin/doc?formal/00-05-02, May 2000.

[6] Workflow Management Coalition, "Workflow Standard - Interoperability Abstract Specification", http://www.wfmc.org/standards/docs/TC-1012_Nov_99.pdf, November 1999.

About the author

Conrad Bock is a Computer Scientist at the National Institute of Standards and Technology. He is the workgroup lead for activities and actions in the UML 2 submission team, and can be reached at conrad.bock@nist.gov .

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 5, September-October 2003

UML 2 Activity and Action Models

Part 2: Actions

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

This is the second 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 covered motivation and architecture for the new models, basic aspects of UML 2 activities and actions, and introduced the general notion of behavior in UML 2 [2]. The remainder of the series elaborates specific elements in the models. This article recaps behavior models in UML and the role of actions in them. It covers the execution characteristics of actions in general, which inherit to the many kinds of actions provided in UML 2. It also covers additional characteristics of actions that invoke behaviors.

1 UML BEHAVIOR MODELS

UML is divided into structural and behavioral specifications, that is, models of the static and dynamic aspects of a system. Behavior models specify how the structural aspects of a system change over time. UML has three behavior models: activities, state machines, and interactions. Activities focus on the sequence, conditions, and inputs and outputs for invoking other behaviors, state machines show how events cause changes of object state and invoke other behaviors, and interactions describe message-passing between objects that causes invocation of other behaviors. Each kind of behavior model emphasizes a different aspect of system dynamics, making one or the other more suitable for a particular application, or stage of application development.

Behaviors can be invoked directly, or as methods on objects accessed through operations on objects. For example, in Figure 1 the DELIVER MAIL behavior can be invoked directly, or through the DELIVERMAIL operation on an instance of the POEMPLOYEE class (the arrow is only for exposition, it is not UML notation). Invoking a behavior through an operation means that an object determines at runtime which behavior actually executes, whereas no object is needed to invoke a behavior directly and the behavior to be invoked is specified at design-time. The provision for first-class behaviors independent of objects was introduced in UML 1.5 [3]. These can be used, for example, when modeling function libraries of popular programming languages, They also facilitate application of UML by modelers who do not use object-orientation (OO) routinely, such as system engineers and enterprise modelers, and provides them a path to incrementally adopt OO as needed. The flexibility to combine OO with functional approaches considerably widens and integrates the potential applications of UML.

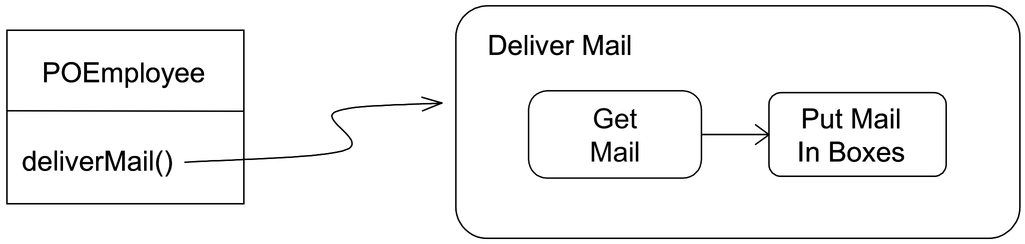


Figure 1: Operation Using a Behavior as a Method

2 THE ROLE OF ACTIONS IN UML

The recursion of behaviors coordinating other behaviors described in the last section is mediated by, and bottoms out at, actions. Actions are the only elements in UML that can query objects, have a persistent effect on them, invoke operations on them, and invoke behaviors directly. For this reason, actions are sometimes called the "primitive" dynamic elements in UML, since all behaviors must eventually reduce to actions to have any effect on objects, or even to invoke other behaviors. Before the introduction of a complete action model in UML 1.5, users depended on platform- or vendor-dependent code inserted in the model to represent queries and effects on objects, and invoke behaviors. UML 1.5 was the first version to support complete behavior specification independently of implementation.

Actions are not behaviors themselves, because actions are provided by UML whereas behaviors are user-defined. UML defines actions for invoking behaviors, either directly or through an operation. For example, in Figure 1 the smaller, round-cornered rectangles are actions that invoke the user-defined behaviors GET MAIL and PUT MAIL IN BOXES. Other actions are defined for getting the values of attributes, linking objects together, and so on. In fact, UML defines enough actions for almost all applications.

Actions are directly contained only in activities. Other specification elements refer to actions through activities. For example, an activity can be used as the method for an operation on a class, as the entry or exit behavior on a state, as a behavior nested inside another activity, and as the cause of a message being sent between objects on interaction lifelines.[1] Actions are a kind of node in activities, connected to other nodes by edges to

[1] The use of activities by interactions is unfortunate for maintaining consistency between interactions and the other behavior models. Interactions are filtered views that show only the message-passing aspects of a behavior. Interactions should refer to actions that pass messages, not to activities containing the actions, so the interaction can show the message view of an activity or state machine. In UML 1.4 and earlier, interaction messages referred to actions.

form a complete flow graph. The two other kinds of nodes in activities mediate between actions to determine when the actions are executed and what inputs they will have. Control nodes route values through the graph, including constructs for choosing between alternative flows, for proceeding along multiple flows in parallel, and so on. Object nodes hold data values temporarily as they wait to move through the graph, including constructs for holding inputs and outputs of actions (see next section).[2]

Actions are notated with round-cornered rectangles, as shown in Figure 2. The wording inside the action is not normative, because a standard textual notation for actions is not adopted yet. Also action nodes can be given labels that are more descriptive for the modeler than the predefined action name in UML. For example, the predefined CREATEOBJECTACTION in UML can be used to instantiate any class, and may be used in business application to create orders. The action might be named CREATE ORDER for clarity. The round-cornered rectangle notation is overloaded in UML, because it is also used by state machines to notate states, which have a very different meaning. This is not a desirable situation, but it is beneficial to those users and vendors who have been trying to apply state machines to flow modeling, for lack of a flow modeling standard until now.

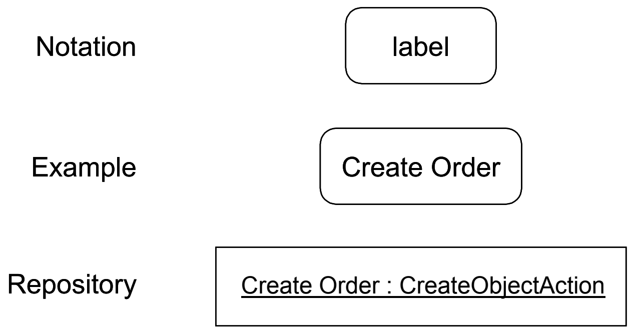


Figure 2: Action Notation and Repository

3 START AND END CONDITIONS FOR ACTIONS

To begin executing, an action must know when to start and what its inputs are. These conditions are called control and data, respectively.[3] Figure 3 shows control and data flow edges in UML 2 directed towards an action (the sources of the flows are not shown). Data flow is distinguished from control by small rectangles on the action to show the type of data flowing into the action. These are called *pins*. Pins may also be notated as a single rectangle standing apart from the action, as long as the source of the data flow provides the same type as the targeted input requires. In Figure 3, the ACQUIRING COMPANY input is shown this way. The meaning is exactly the same regardless of which notation is used for pins, as is the repository model. See Figures 5 and 6 in the first article.

[2] Contrary to the name, object nodes can hold both objects and data. UML unifies data and object under the notion of classifier.

[3] Data includes objects, see footnote 2.

Pins provide for multiple data flows into an action, which may all be of the same type but treated differently by the action. For example, the ACQUISITION behavior in Figure 3 has two data inputs, both of type COMPANY, where one is the purchasing company and the other is purchased. A pin can be labeled with any string, but it is usually the name of the type of object being input or output, such as COMPANY, or where that is ambiguous, a distinguishing name like ACQUIRED COMPANY, or both separated by a colon, as in COMPANY : ACQUIRED COMPANY. Pins are kinds of object nodes: they hold inputs to actions until the action starts, and hold the outputs of actions before the values move downstream.

An action begins executing when all its incoming control and data are available (see exceptions to this in section 5). For example, the ACQUISITION behavior in Figure 3 will only start when the two companies and control arrive at the action. If there are multiple control flow edges coming into an action, control must arrive along all of them for the action to start. For convenience, an action with no incoming control starts when all its data inputs are available. This simplifies diagrams where the control dependencies between actions in an activity are exactly the same as the data dependencies. An action with no incoming control or data will never execute, though there are some exceptions provided for convenience, described later in the series.

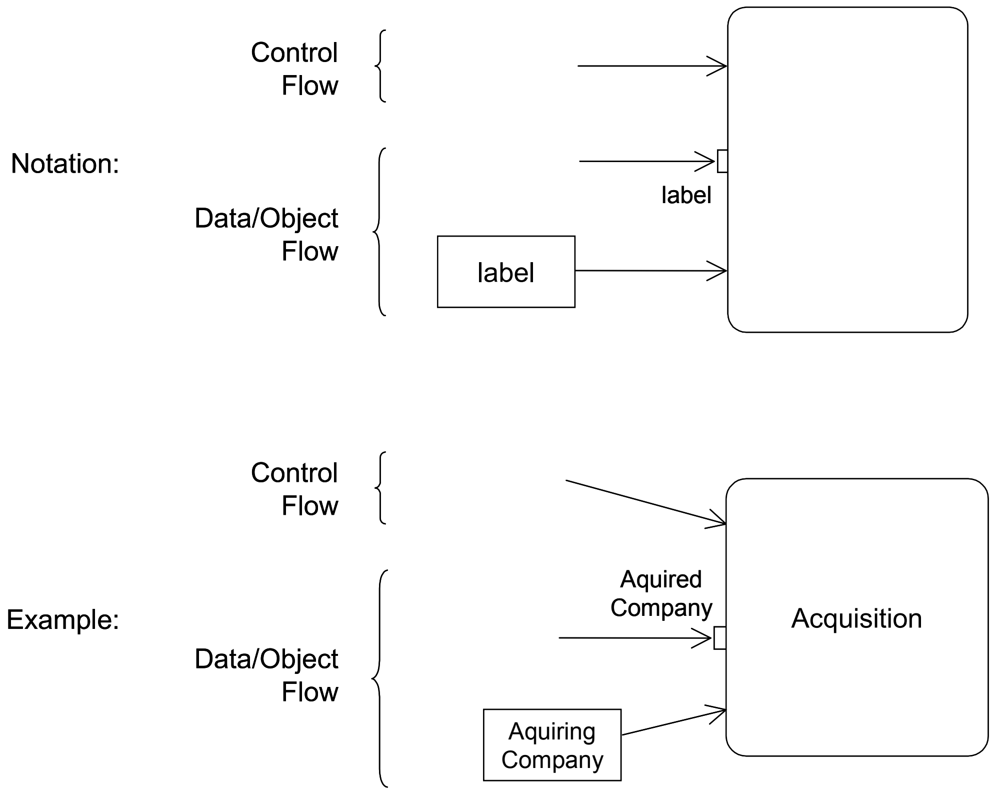


Figure 3: Control and Data Flow into an Action

Control flow does not require pins because control is not operated on by actions, that is, actions cannot take a control value as input, examine it, perform some operation on it, as they can do with data. Consequently, control does not have a type and is not a form of data. There is in effect only one control value, namely the one that indicates an action can start executing. Even though control flow has no pins, control arriving at an action before other inputs is still held until those other inputs arrive before the action starts. However, there is no provision for holding multiple control values as there is for data. The topic of queuing will be addressed later in the series.[4]

Start conditions for actions treat data as a form of control in the sense that the availability of data can trigger the start of an action, just as control can. This contrasts with other forms of data flow in which data is a passive element that is read by an action as needed when control indicated the action should start [4][5]. Passive data, usually called data store, is partially addressed in UML 2 as a kind of object node, described later in the series.

Modelers cannot change the fact that all inputs are required for an action to start. This makes activity diagrams easier to read because the way actions start is uniform, rather than varying by action, requiring detailed annotations to show the differences. Exceptions to this are restricted to a few common patterns, as explained in section 5. User-defined variations in control and data combination are separated from actions and explicitly shown by using control nodes. This contrasts with languages that provide for variations or user-defined control and data combiners at each action to specify other start conditions [6][7][8]. Such languages can result in notations that are difficult to interpret, because the execution rules for actions are not uniform.

Input pins accept data and objects at runtime, while other information that actions require to begin execution is provided statically in the user model, at design time. For example, the action in Figure 4 sets an attribute value. At runtime the input pins receive the object to be modified, and the value to be added to the attribute. The attribute itself is constant for all executions of the action and does not have a corresponding pin. It is only shown informally in the action name. The attribute appears in the repository model for the action, shown in Figure 5, as the STRUCTURALFEATURE association from the ADDSTRUCTUREALFEATUREVALUEACTION to PROPERTY (see [9] regarding the UML repository). A tool can access the repository to show the attribute in a vendor-specific way, which may be on the diagram, for example as a vendor-specific action naming convention, or may be some other non-diagrammatic user interface technique, such as a dialog box.

[4] Some languages provide for treating control as data, including queuing of control, as called for in [8].

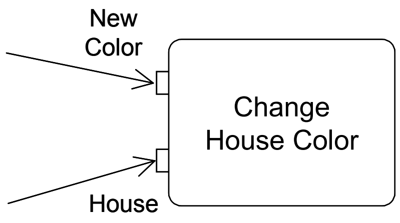


Figure 4: Action for Setting an Attribute Value

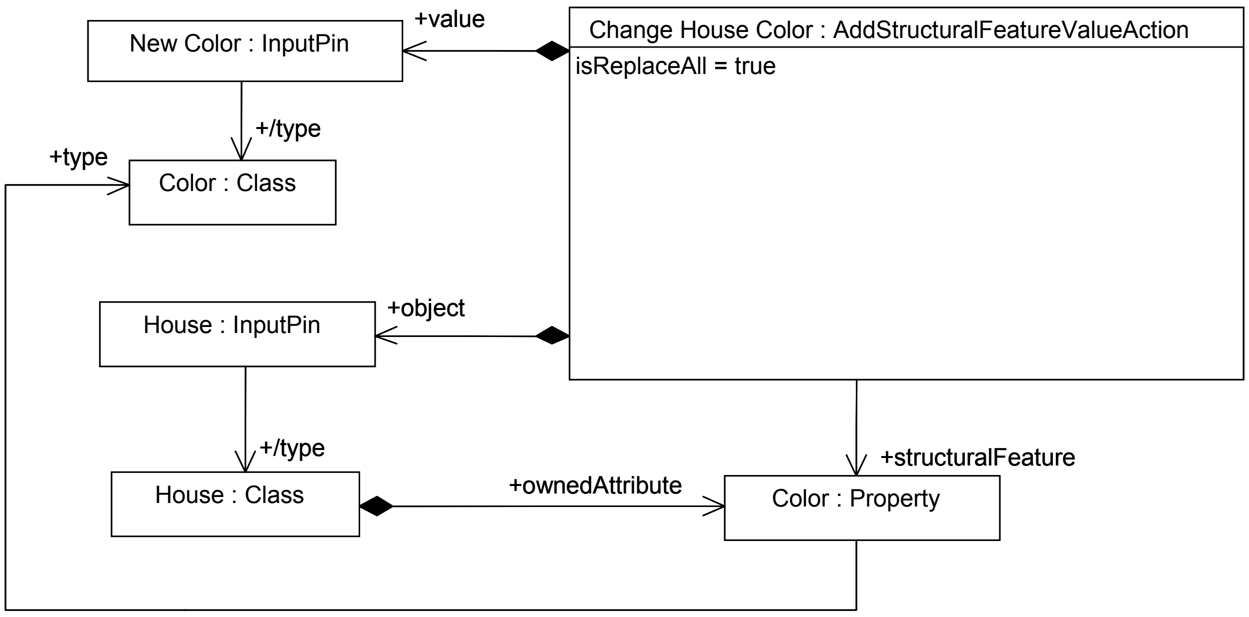


Figure 5: Repository Model for Figure 4

The repository model also includes the derived values for the type of object that pins can hold, called the pin type for short. For example, in Figure 5 the type of the VALUE pin, COLOR, is derived from the attribute being modified by the action, as is the type of the OBJECT pin, HOUSE. The derivation rules for pin types of each action are given by constraints in the UML specification.

An action has control and data outputs, notated in the same way as inputs, except the flow arrows point in the other direction, as shown in. An action terminates based on conditions internal to itself, but when the action does terminate, data is posted to all its output pins, and control values are placed on all its outgoing control flows (see exceptions to this in section 5). For example, in, when ACQUISITION terminates, control leaves the action along with the merged company and new shares to replace the acquired ones. An action that has no control or data outputs can still terminate, but its termination cannot cause other actions to start.

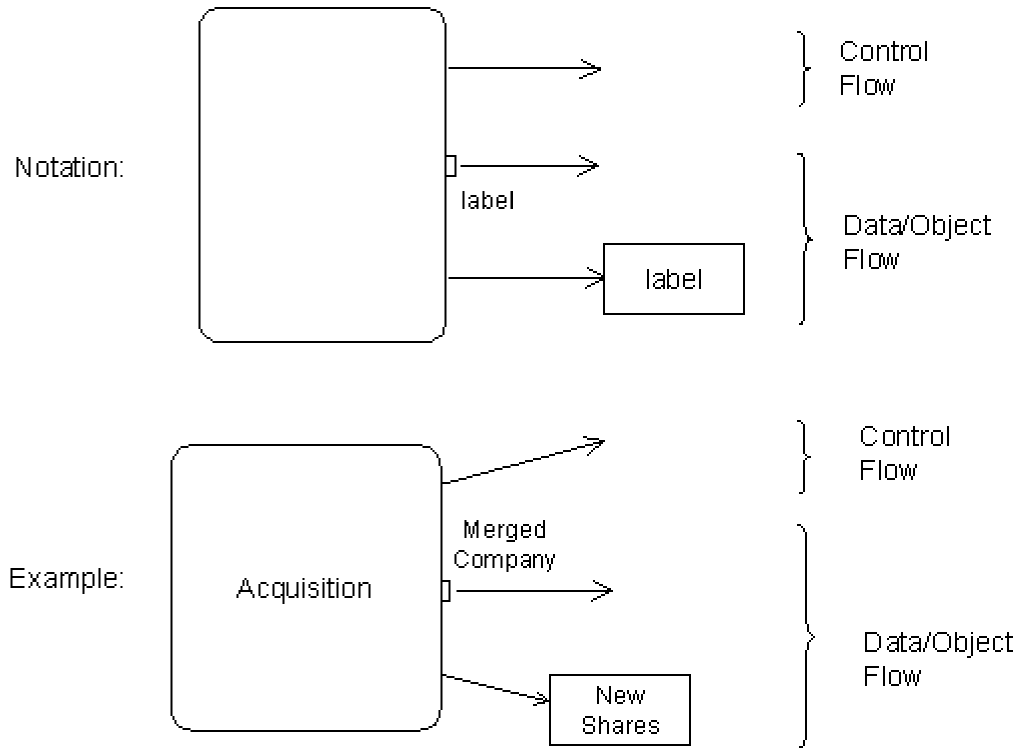


Figure 6: Control and Data Flow out of an Action

4 PINS, PARAMETERS, AND CALL ACTIONS

Actions are predefined in UML, whereas behaviors and operations are user-defined. Modelers can specify behaviors as activities, state machines, or interactions, can use behaviors as methods for operations on classes, and can invoke behaviors and operations with the actions CALLBEHAVIORACTION and CALLOPERATIONACTION, respectively.[5] Figure 7 shows an example of these actions, collectively known as call actions. The behavior DELIVER MAIL BY TRUCK is invoked by a CALLBEHAVIORACTION in the fragment at the top, while the operation DELIVER MAIL on an instance of POST OFFICE is invoked by CALLOPERATIONACTION in the fragment at the bottom (the curved line is

[5] Behaviors can also be invoked by sending signals to objects with SENDSIGNALACTION, BROADCASTSIGNALACTION, and SENDOBJECTACTION, but this is usually indirect invocation through triggering transitions on a state machine of the object. The responding behavior can vary over time even for the same object due to multiple states, can be delayed due to the queuing of signals for processing by the state machine, and may even not have inputs compatible with the signal that was sent. The discussion in sections 4 and 5 does not apply to these cases, because the loose linkage of action and behavior means the characteristics of the parameters cannot be notated on the pins. Likewise for operation calls that are dispatched to a state machine, which is also possible in UML. For operations and signals that are defined to uniformly and directly invoke a behavior, the discussion in sections 4 and 5 applies.

only for exposition, it is not UML notation).[6] The behavior DELIVER MAIL BY TRUCK can be a method for the DELIVER MAIL operation on POST OFFICE. The repository model for this case is shown in Figure 8.[7] The repository for CALLBEHAVIORACTION on the DELIVER MAIL BY TRUCK behavior is very similar, except that CALLBEHAVIORACTION : ACTION is linked directly to DELIVER MAIL BY TRUCK : BEHAVIOR, rather than going through an operation.

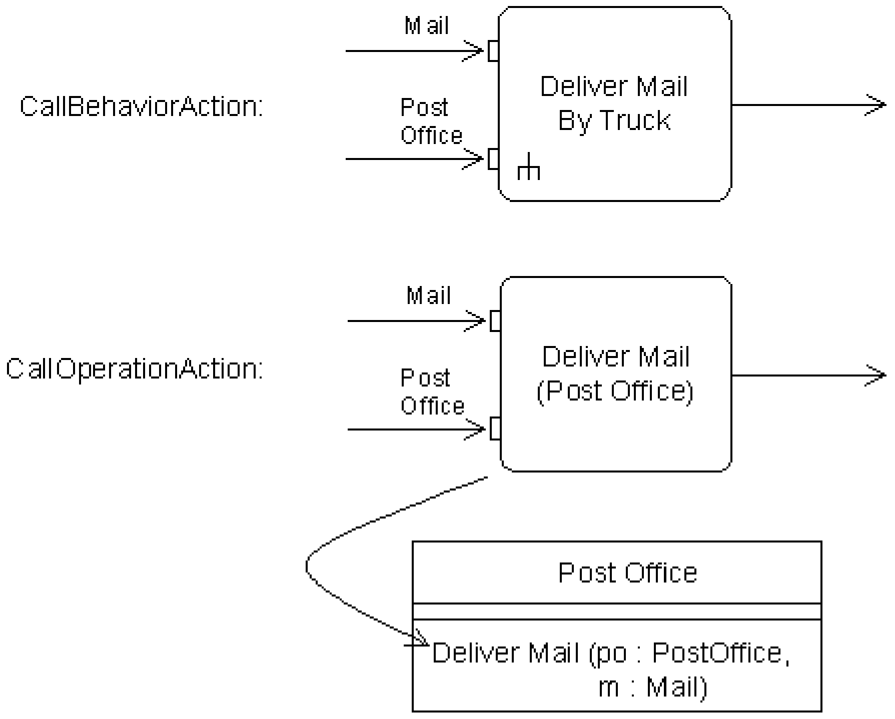


Figure 7: Examples of CALLBEHAVIORACTION and CALLOPERATIONACTION

Behaviors and operations have parameters, and these must be compatible with each other when a behavior is used as the method for an operation, as DELIVER MAIL BY TRUCK and DELIVER MAIL are. Parameters must also be compatible with pins of actions that invoke the behaviors or operations, as the pins are in Figure 7.[8, 9] Labels on pins for call actions

[6] The rake symbol for CALLBEHAVIORACTION was under some debate at the time of adoption and may be changed in finalization. The parentheses notation for CALLOPERATIONACTION is inherited from UML 1.x, but extensions to it in UML 2 for showing a different operation name than the action name may also change in finalization. There is no notation yet to indicate which pin provides the object on which the operation is invoked (for the "self " parameter), when it isn't clear from the class given in parentheses.

[7] The DELIVER MAIL operation is polymorphic even though POST OFFICE has only one method for it. Subtypes of POST OFFICE may override the method with others, and CALLOPERATIONACTION will dispatch to whatever method is in place for the class of object that is the target of the action.

[8] Pins and parameters may be incompatible only when the call action is asynchronous that is, does not wait for the return values after invoking the behavior or operation. In this case, the out and return parameters of the behavior or operation, if any, will have no corresponding pins. Only the in and inout parameters must be compatible.

can use the full UML syntax for parameters as they appear in classes, which includes the parameter name, type, direction, and so on. For example, the pin for the mail on DELIVER MAIL could be labeled as "IN M : MAIL".

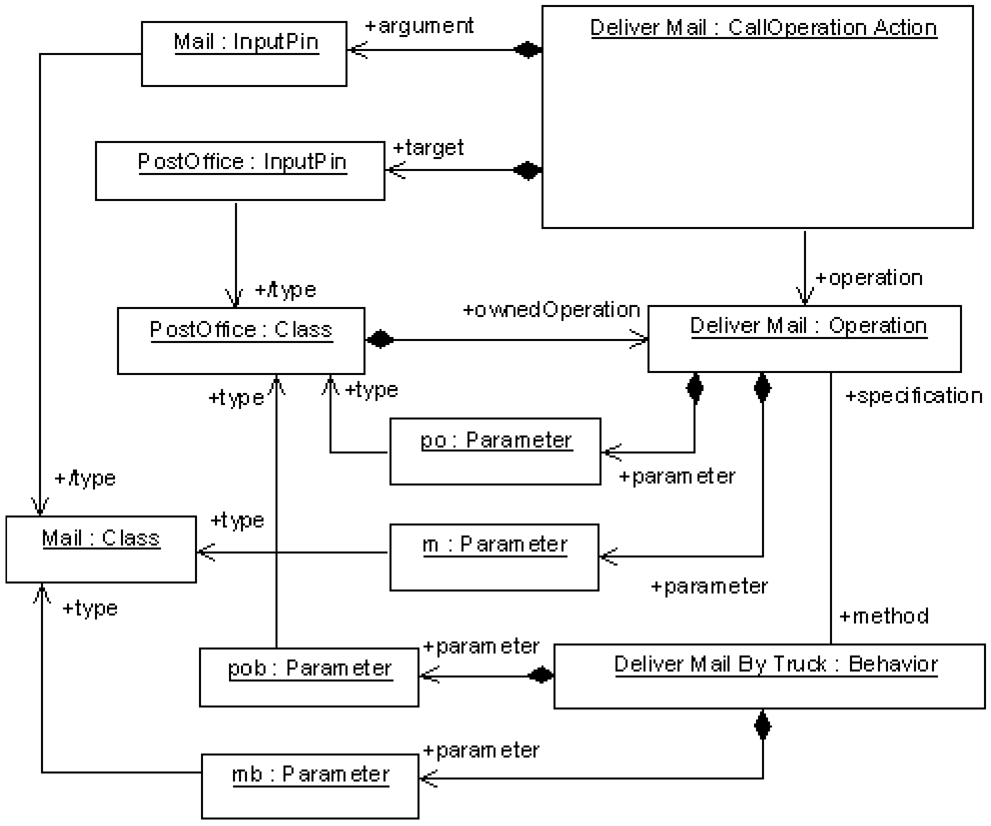


Figure 8: Repository Model for CALLOPERATIONACTION in Figure 7.

Pins on call actions are usages of parameters in the same way that call actions are usages of behavior and operations. Pins and actions separate the potentially reusable specification of a behavior or operation from actual usage, to support multiple usages of the same specification in multiple behaviors. For example, the same DELIVER MAIL behavior may be invoked in many activity diagrams, or many times in the same activity diagram, but each invocation will be represented by a separate instance of CALLOPERATIONACTION in the repository, and separate instances of INPUTPIN and OUTPUTPIN, all referring to the same DELIVER MAIL operation and parameters. This is because each usage of DELIVER MAIL and its parameters will be in a different flow graph, or at different points in the same flow, and the flow edges from the action and pins will

[9] The matching of pin to parameter is achieved by the fact that the links from actions to input and output pins are ordered, and the links between parameter and behavior or operation are also ordered, so can the mapped one-for-one. However, the UML 2 currently uses separate associations from ACTION to INPUTPIN and OUTPUTPIN, while there is a single association from BEHAVIOR to PARAMETER and from OPERATION to PARAMETER. UML must either define a convention for combining the pin links into a total order, or add a single generalized association between ACTION and PIN.

come from and go to different actions in each case. Without this separation, the reusable specification of DELIVER MAIL would become embedded in all the other behaviors that use it, and have all the flow links of all its usages tied to it.

5 STREAMING AND EXCEPTION PARAMETERS, PARAMETER SETS

Call actions can have different start and end conditions than the normal ones described in section 3. These conditions depend on characteristics of the parameters of the behavior or operation being invoked. They are:

1. **Streaming Parameters**
   Streaming parameters can accept or provide values while an action is executing. They are not required to be input only when the action begins execution or output only when it stops, as non-streaming parameters are. In Figure 9, for example, GENERATE LEADS can continue to execute as it outputs leads for FOLLOW LEADS. Likewise, FOLLOW LEADS can accept new leads as it processes others.[10] Streaming parameters are indicated with the annotation {STREAM} near the pin corresponding to the parameter, or in the alternate notations shown in the lower parts of Figure 9.

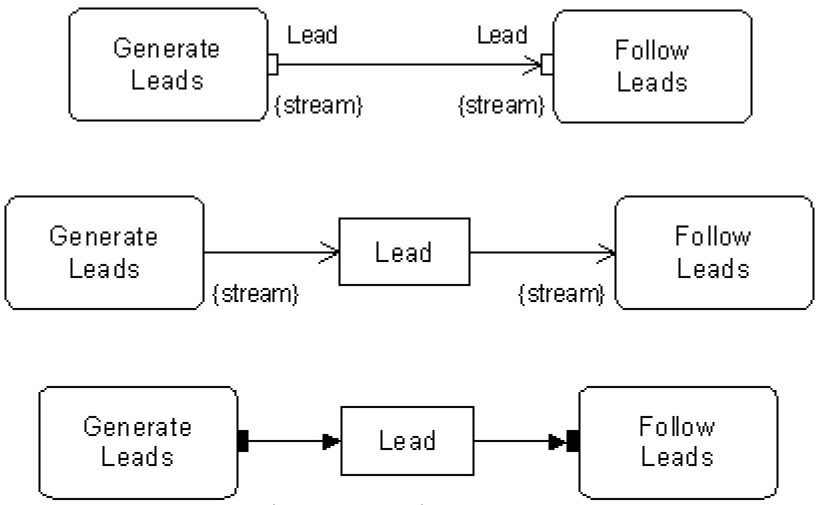


Figure 9: Streaming Parameters

[10] Streaming input parameters are currently required to accept at least one value before the action can stop executing. That is, streaming inputs values are not optional, they are just allowed to arrive after the execution begins. This is to address the problem of an input arriving after the execution is done and being consumed by a later execution with other later inputs it should not be paired with. This rule does not actually work very well, since multiple inputs may arrive after the execution is done that the execution also should have waited for. Streaming output parameters on the other hand, are not required to provide values at all for the action to terminate. It is proposed for future revisions of UML to separate streaming from optionality, and even multi-valuedness, so these concepts can be used independently.

Streaming parameters can accept or provide multiple values during a single behavior execution (this is where the name "stream" comes from, even though multiple values are only allowed, not required). Modeling streams usually involves multiple values flowing in a single activity. For example, if FOLLOW LEADS were defined as an activity, multiple leads would be flowing through the graph at one time. Multi-token activities will be covered later in the series.

2. **Exception Output Parameters**
Exception output parameters provide values to the exclusion of any other output parameter or outgoing control of the action. For example, in Figure 10 FILLORDER normally outputs a package for shipping, paperwork for accounting, and a control flow for the next step in the business process. It also has an exception parameter notated with a triangle near the corresponding pin, which is output when the order is incompletely specified. If the exception output is provided, the other outputs and outgoing control are not, and the action must immediately terminate. This means the normal business process no longer occurs. See comparison to parameter sets below.[11]

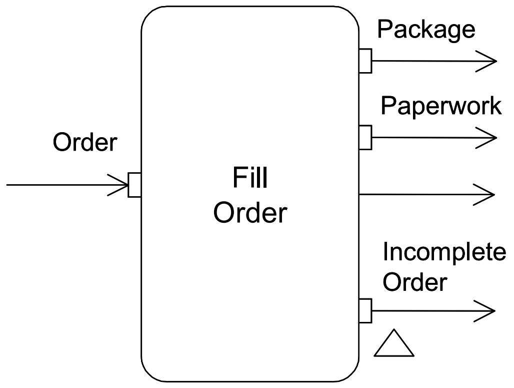


Figure 10: Exception Output Parameter

There can be multiple exception output parameters, but at most one can provide a value per action execution. Streaming output values posted before an exception output are not affected by the exception, because they have already flowed out of the action.

3. **Parameter Sets**
Parameters can be grouped so that exactly one of the groups can accept or provide values for the action. For example, in Figure 11, the FILLORDER either outputs a package and paperwork, or a partially completed product and incomplete order notice. This could happen if FILL ORDER starts assembling the product without

[11] Exception output parameters can be combined with streaming input parameters to model interruption. A behavior might define a streaming input parameter that it reacts to by immediately outputting an exception and terminating.

checking the order first, and produces a partial product, returning it until the order is completed. In contrast, FILL ORDER in Figure 10 expects the order to be checked ahead of time, because it outputs an exception if it is incomplete. See the next section for how to declare modeler expectations on actions.

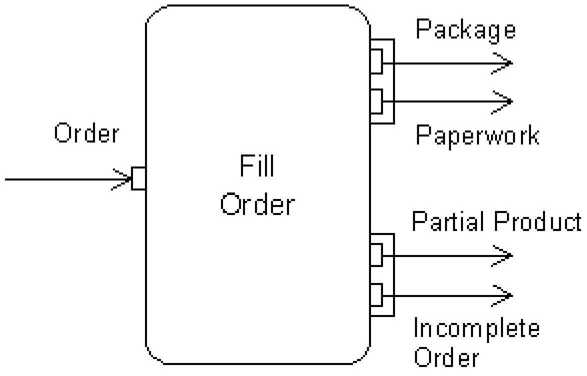


Figure 11: Parameter Sets

Parameter sets must have at least one parameter in them, but the same parameters can be part of more than one parameter set, to model parameters that always accept or provide values regardless of which parameter set is used by an execution.

Exception parameters could be considered a special form of parameter set where each exception parameter is in its own set and all the non-exception parameters are in another set. However, one difference is that behavior execution is immediately terminated when the exception output is posted. For example, in Figure 10, if FILL ORDER is a CALLBEHAVIORACTION invoking another activity, the posting of an incomplete order will stop any other values flowing in that activity. Also, parameter sets cannot output control values, because they are groups of pins and control is not output on pins (see section 3). Exceptions will be covered in more detail later in the series.

The above characteristics of parameters apply to operations on objects as well as behaviors. When an operation is invoked on an object, a behavior is chosen with parameters compatible to the operation, including the above characteristics. Once the behavior is chosen, the semantics of invoking it is the same as if the behavior was invoked directly, without dispatch from an object.[12] The notation in class diagrams for the above parameters is the property list. If the behaviors in the above figures were operations, they would appear in class diagrams as (omitting the "self" parameter):

[12] In theory, different behaviors could be chosen each time the operation is called in the same runtime object, but in practice, it is usually the same behavior each time. In rare implementations that allow methods to change at runtime, the implementation will not support streaming inputs, because these require an input arriving after execution has started to be forwarded to the existing execution, which is not unique in such implementations.

- Figure 9: `FollowLeads (in i: Lead {stream})`

- Figure 10:
```
FillOder (in  o   : Order,
          out pkg : Package,
          out pw  : Paperwork,
          out io  : IncompleteOrder {exception})
```

- Figure 11:
```
FillOder(in  o   : Order,
         out pkg : Package   {parameterSet ps1},
         out pw  : Paperwork {parameterSet ps1},
         out pp: PartialProduct {parameterSet ps2},
         out io: IncompleteOrder{parameterSet ps2})
```

It is not required that the diagram show all the above information. For example, the parameter set names are not shown in Figure 11.

In summary, the start and end conditions for call actions are:

- All non-stream inputs must arrive for the action to begin. If there are only stream inputs, then at least one must arrive for the behavior to begin.

- All inputs must arrive for the behavior to finish, that is, all inputs must arrive before non-stream and control outputs can be posted. This includes streaming inputs. No input is optional.

- Either all non-stream, non-exception outputs must be provided when an activity is finished, or one of the exception outputs must be, but not both. Exception outputs cannot be streaming, because the action terminates when an exception output is posted.

- A behavior or operation with input parameter sets can only accept inputs from parameters in one of the sets per execution. A behavior or operation with output parameter sets can only provide outputs to the parameters in one of the sets per execution. The start and end conditions in the previous bullets apply to each set separately.

The above deviations from normal start and end conditions described in section 3 are ways that an action can affect the routing of values through the graph that contains the action. Normally only control nodes and edges in an activity can determine which way value flows throw the graph, and actions just operate on the values routed to them, and hand the values back to the graph for further routing. The additional parameter characteristics above are useful in situations where routing decisions should be encapsulated in a behavior, to be reused by various actions. Otherwise, the decisions would need to be reproduced across control nodes and edges in multiple activities. For example, in Figure 10, the test for incomplete orders would need to be remodeled in a decision node after every invocation of FILLORDER, instead of being defined once in the FILLORDER behavior.

6 LOCAL PRECONDITIONS AND POSTCONDITIONS

Modelers can declare conditions before and after actions that the rest of the model is expected to insure are actually true. These can be the familiar pre/postconditions on operations and behaviors [10], discussed later in the series, or can be local to specific actions, including individual invocations of behaviors or operations. For example, Figure 12 shows an action that dispenses a drink from a vending machine in the context of an activity for dieting. The local precondition is that the drink requested is low in calories. The expectation is that the rest of the activity around DISPENSE DRINK will ensure that the name provided is actually for a low-calorie drink.

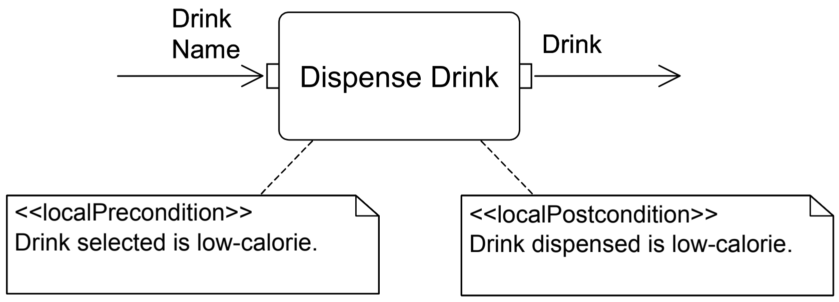


Figure 12: Local Preconditions and Postconditions

The above conditions are called "local" because they only apply to the individual action, not to all the invocations of the behavior or operation. For example, the DISPENSE DRINK behavior could be used in activities unrelated to dieting. It is only restricted to low-calorie drinks at the particular call action in Figure 12.[13] A global precondition on the DISPENSE DRINK behavior might be that the drink requested is available from the machine. This would apply to every invocation of DISPENSE DRINK in every activity that uses it.

Since pre/postconditions are inherently redundant with the rest of the model, UML intentionally does not dictate when or whether pre/postconditions are tested.[14] For example, they may be tested by a program correctness verifier at design time, or during compilation of the model to a specific platform, at runtime during the execution of the activity, or not at all when out of debugging mode, for example. UML also does not define what the runtime effect of a failed pre/postcondition should be. For example, it may be an error that stops execution, or just gives a warning, records a log entry, or has

[13] UML could in theory provide for local pre/postconditions at every kind of node and edge in the activity flow, such as control and object nodes. UML 2.0 happens to support them on actions only.

[14] Perhaps future versions of UML should provide for modeling the runtime effect of local pre/postconditions and constraints generally.

no effect at all. And as usual, UML does not specify an implementation. Pre/postconditions can be implemented as assertions in programming languages, but it is not required.[15]

7 CONCLUSION

This is the second in a series on the UML 2 activity and action models. This article focuses on actions as the basis for all the behavior models in UML, and the execution characteristics of actions that are common to all the specific actions provided in UML 2. Static and dynamic inputs are distinguished, and start and end conditions for actions are given in terms of runtime inputs and outputs of the action. Default start and end conditions are described, along with special capabilities for actions that invoke behaviors. The relation between pins and parameters is discussed in this context.

ACKNOWLEDGEMENTS

Thanks to Evan Wallace and James Odell for their input to this article.

[15] Since pre/postconditions are modeler-defined constraints with no standard effects, violations do not mean that the semantics of the action is undefined as far as UML goes. Violations only mean the model or execution trace does not conform to the modeler's intention, and may reach points later on where execution aborts or throws exceptions due to conditions that the modeler did not expect.

REFERENCES

[1] U2 Partners, "Unified Modeling Language: Superstructure," version 2.0, 3rd revised submission to OMG RFP ad/00-09-02, http://www.omg.org/cgi-bin/doc?ad/2003-04-01, April 2003.

[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

[3] Object Management Group, "OMG Unified Modeling Language, version 1.5," http://www.omg.org/cgi-bin/doc?formal/03-03-01, March 2003.

[4] Rumbaugh, J., et al., *Object-oriented Modeling and Design*, Prentice Hall, 1991.

[5] Shlaer, S., Mellor S., *Object-oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

[6] Thatte, S., et al., "Business Process Execution Language for Web Services," http://www-106.ibm.com/developerworks/library/ws-bpel/, May 2003.

[7] Workflow Mangement Coalition, "Workflow Process Definition Interface," http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf, October, 2002.

[8] OMG Systems Engineering DSIG, "UML for Systems Engineering RFP," http://www.omg.org/cgi-bin/doc?ad/03-03-41, March 2003.

[9] Bock, C., "UML Without Pictures," to appear in IEEE Software Special Issue on Model-driven Development, September/October 2003.

[10] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, 1997.

About the author

Conrad Bock is a Computer Scientist at the National Institute of Standards and Technology, specializing in process models. He is one of the authors of UML 2 activities and actions, and can be reached at conrad.bock at nist.gov .

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 6, November-December 2003

UML 2 Activity and Action Models

Part 3: Control Nodes

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

This is the third 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 previous article addressed the execution characteristics of actions in general, and additional functionality of actions that invoke behaviors [2]. The first article gave an overview of activities and actions that is assumed here [3]. The remainder of the series elaborates other specific elements. This article covers control nodes, which route control and data through the flow model. It also points out the differences in concurrency support between UML 2 and UML 1.x activities.

1 CONTROL NODES

To recap, UML 2 activities contain nodes connected by edges to form a complete flow graph. Control and data values flow along the edges and are operated on by the nodes, routed to other nodes, or stored temporarily. More specifically, action nodes operate on control and data they receive via edges of the graph, and provide control and data to other actions; control nodes route control and data through the graph; and object nodes hold data temporarily as they wait to move through the graph. Data and object are unified in UML under the notion of classifier, so they are used interchangeably. The term "token" is shorthand for control and data values that flow through an activity.

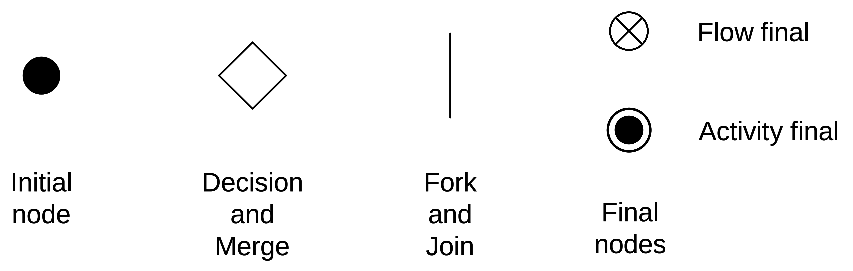


Figure 1: Control Nodes

There are seven kinds of control node, with five notations, as shown in Figure 1. Contrary to the name, control nodes route both control and data/object flow. Each of them is described in the sections below.

2 INITIAL NODES

Flow in an activity starts at initial nodes. They receive control when an activity is started and pass it immediately along their outgoing edges. No other behavior is associated with initial nodes in UML. Initial nodes cannot have edges coming into them. For example, in Figure 2, when the DELIVER MAIL activity is started, a control token is placed on the initial node, notated as a filled circle, and immediately flows along to start the GET MAIL action.

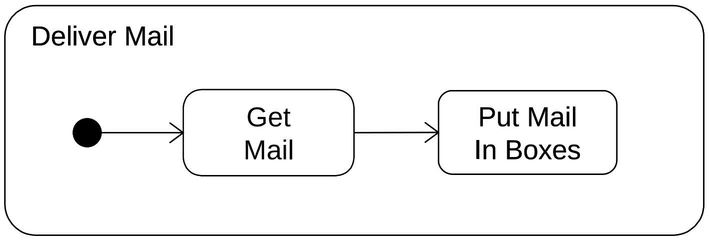


Figure 2: Initial Node

An activity can contain more than one initial node. A single control token is placed in each one when the activity is started, initiating multiple flows. It might be clearer to use one initial node connected to a fork node to initiate multiple flows simultaneously (see section 5), but this is up to the modeler. Other ways to start flows in an activity will be discussed later in the series.

If an initial node has more than one outgoing edge, only one of the edges will receive control, because initial nodes cannot copy tokens as forks can (see section 5). In principle, the edges coming out of initial nodes can have guards and the semantics will be identical to a decision node (see next section). For convenience, initial nodes are excepted from the general rule that control nodes cannot hold tokens waiting to move downstream, if it happens that all the guards fail. In general, it is clearer to use explicit decision points and object nodes than to depend on these fine points of initial nodes.

3 DECISION NODES

Decision nodes guide flow in one direction or another, but exactly which direction is determined at runtime by edges coming out of the node. Usually edges from decision nodes have guards, which are Boolean value specifications evaluated at runtime to determine if control and data can pass along the edge. The guards are evaluated for each individual control and data token arriving at the decision node to determine exactly one edge the token will traverse. For example, Figure 3 shows a decision node, notated as a diamond, choosing between flows depending on whether an order can be filled or not. Value specifications in UML 2 are often just strings interpreted in an implementation-dependent way[1]. In this example, the modeler's intention for the strings "accepted" and "rejected" must already be understood by the implementation or defined by additional modeling. Model refinement can introduce an additional explicit behavior, such as a decision input behavior, explained below.

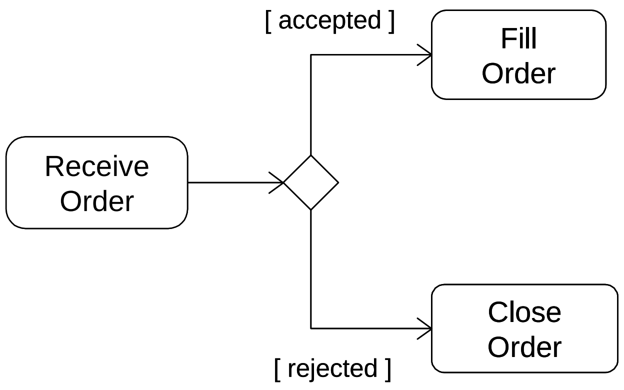


Figure 3: Decision Node

The order in which the above guards are evaluated is not constrained by UML, and can even be evaluated concurrently. For this reason, guards should not have side effects, to prevent implementation-dependent interactions between them. If guards are to be evaluated in order, as is typical in conditional programming constructs, then decision nodes can be chained together, one for each guard, combined with the predefined guard "else". The else guard can be used with decision nodes for a single outgoing edge to indicate that it should be traversed if all the other guards from the decision node fail.

[1] UML 1.x more forthrightly called these "uninterpreted strings", but UML 2 value specifications can also be instance specifications, and opaque or structured. Activities use value specifications in some places and behaviors in others. Whether this is done by any consistent rationale will be addressed in finalization.

Figure 4 shows an example of chained decision nodes with else guards. The CLOSE ORDER action is reached by failure of the non-else guards[2].

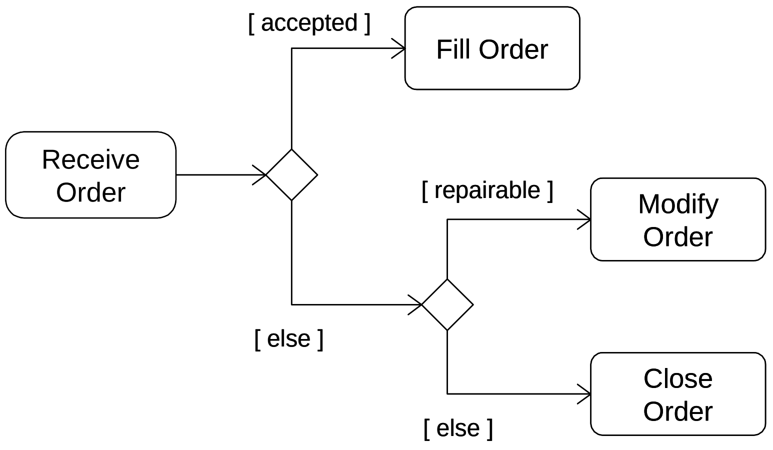


Figure 4: Chained Decision Nodes

Since guard evaluation order is implementation-dependent, the modeler should also arrange that only one guard succeed, otherwise there will be race conditions among them. It is up to the implementation whether to finish evaluating guards after one is found that succeeds. In theory, all the guards might succeed at one time, in which case the semantics is not defined[3]. If all the guards fail, then the failing control or data token remains at the object node it originally came from, since control nodes cannot hold tokens waiting to move downstream, as object nodes can. Token queuing is discussed later in the series.

If the guards involve a repeated calculation of the same value, a behavior on the decision node can determine this value once for each token arriving at the decision node, and then provide it to the outgoing guards for testing. For example, Figure 5 shows a decision input behavior IS ORDER ACCEPTABLE providing a Boolean result tested by the outgoing guards[4] (the curved arrow is not part of UML notation, see earlier articles on inputs and outputs of actions and activities). Each order arriving at the decision node is passed to IS ORDER ACCEPTABLE before guards are evaluated on the outgoing edges[5]. The

[2] Conditional constructs can also be modeled with a CONDITIONALNODE. This is one of the aspects of activities for modeling programming language constructs. These will be covered later in the series.
[3] See discussion of undefined semantics in section 6 of the second article [2].
[4] An alternate notation is { decisionInput = Is Order Acceptable } placed near the decision node.
[5] Object flow edges are usually distinguished from control edges by rectangles representing the type of object that is flowing, for example as pins on actions in Figure 5. It is a presentation option in UML to omit these rectangles as in Figure 3, for example if they are obvious to the reader or confusing to subject matter experts, while still storing the model for them in an underlying UML repository. Special views such as this are a way activities support a wide range of the development cycle, from process sketching to executable program specifications. Model refinement is another technique, which refers to multiple models for the same process existing over time, linked in a progression as detail is added. For example, a subject matter expert might draw a diagram like Figure 3 without pins, and a more UML-knowledgeable modeler might

output of the behavior is available to the guards, in an implementation-dependent way, as with all value specifications (see footnote 1). The value specifications in Figure 5 happen to use the name of the output parameter of the decision behavior.

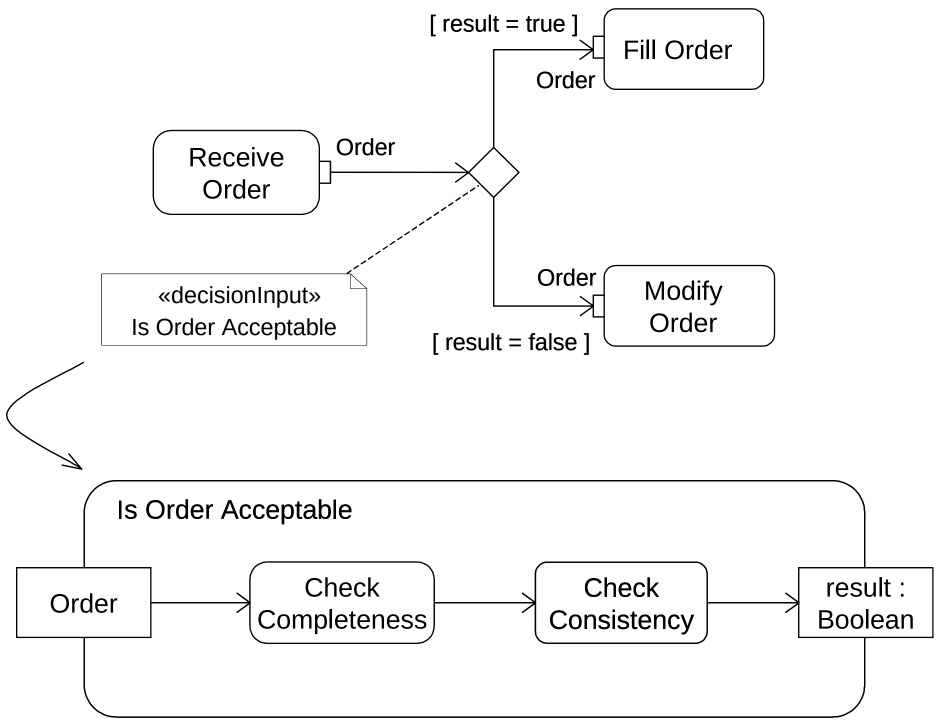


Figure 5: Decision Input Behavior

A repository model for part of Figure 5 is shown in Figure 6 (see first article for more about the UML repository [3]). The two anonymous object flows are separate repository elements for the two object flows coming out of the decision node. Each has an opaque expression as a guard, which are the kind of value specification that are completely implementation-interpreted. Each object flow targets its own separate anonymous input pin, each of which provide input to their respective behaviors, one for each direction of flow from the decision[6].

add them later. A record of refinements can be kept using the upcoming Query, View, and Transformation technology [4]. This is a simple example of the general problem of recording design evolution, to ensure that the end product fulfills the original requirements, as in systems engineering for manufacturing [5].

[6] The current UML specification implies that decision input behaviors only apply to data tokens, but does not explicitly restrict them to that. This is to be clarified in finalization. A decision input behavior for control flow can in principle have no parameters and return a value based on other data, such as available from the host object of the entire activity. The host object is retrieved with the action READSELFACTION. In general, if a behavior requires information that cannot be retrieved from values provided by its input parameters, it can use READSELFACTION. This action is discussed later in the series.

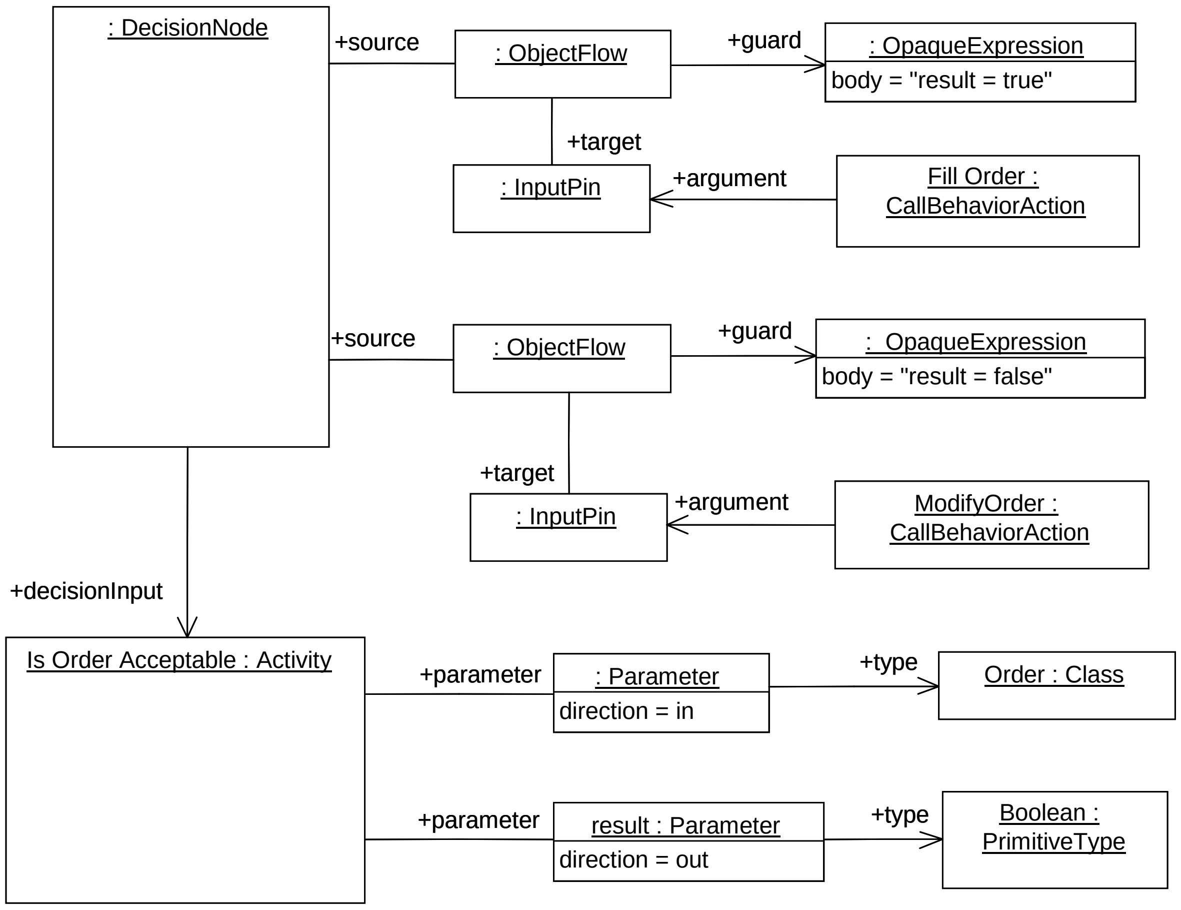


Figure 6: Repository for Part of Figure 5

Other factors besides guards can determine whether control and data can pass along an edge, and consequently which edge will be traversed out nodes, including decision nodes. Future articles will address edges and token queuing in more detail. Whatever factors are involved, the purpose of a decision node is to ensure that each control and data token arriving at the decision node traverses no more than one of the outgoing edges.

4 MERGE NODES

Merge nodes bring together multiple flows. All control and data arriving at a merge node are immediately passed to the edge coming out of the merge. No other behavior is associated with merge nodes in UML[7]. Merge nodes have the same notation as decision nodes, but merges have multiple edges coming in and one going out, whereas it is the opposite for decision nodes. Flows coming into a merge are usually alternatives from an upstream decision node. For example, Figure 7 shows a merge node bringing two flows together to close an order. The merge is required, because if the two flows went directly into CLOSE ORDER, both flows would need to arrive before closing the order, which would never happen [2][8] . Merge can be used with concurrent flows also, see Figure 16 in section 6.

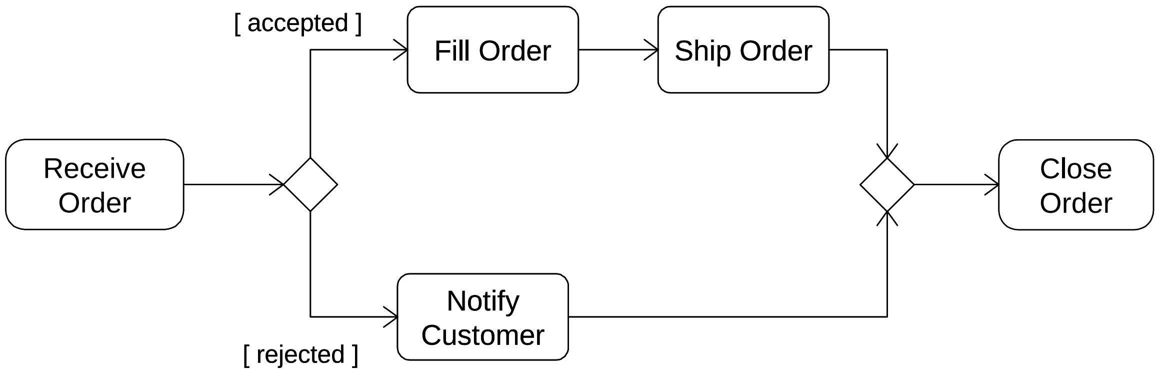


Figure 7: Merge Node with Alternate Flows

Flows from chained decision nodes can be merged more flexibly than with conditional constructs in structured programming languages. For example, Figure 8 shows two of three flows from a decision node being merged separately from the third. Flows coming out of a decision node do not need to be brought together by a merge at all. See Figure 20 in section 7.

[7] Use join nodes for more complex semantics, see section 6.
[8] UML 1.x activities would require only one of the transitions to arrive to start the action, as do all state machines. With UML 2 the example in the UML User Guide, Figure 19-9, is correct, whereas it was incorrect in UML 1.x [6][7].

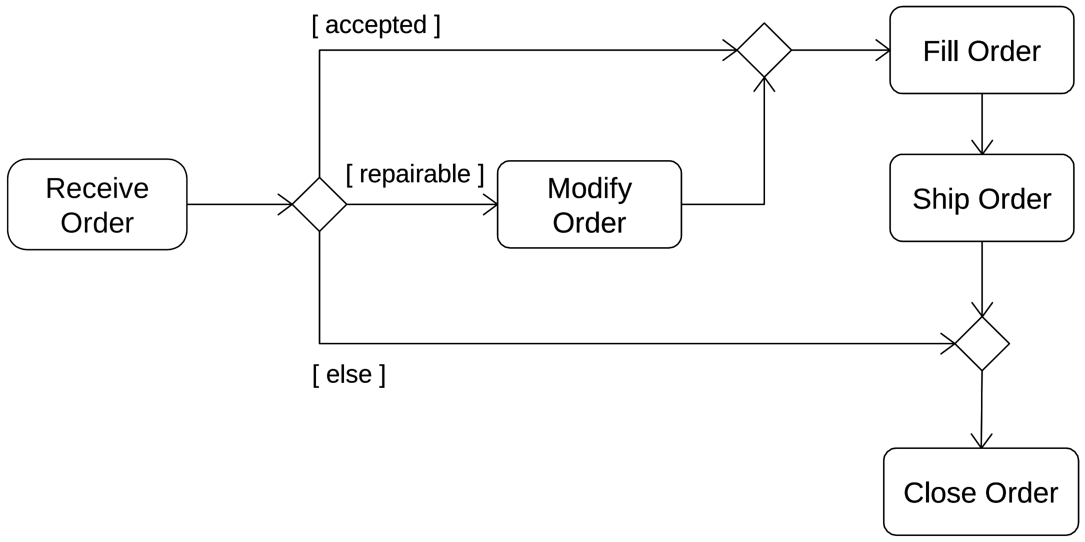


Figure 8: Merge Nodes without Nesting

Figure 9 on the left shows a shorthand notation for a merge immediately followed by a decision. It has the same effect as the separate merge and decision shown on the right. Both have the same repository model, which contains separate merge and decision nodes.

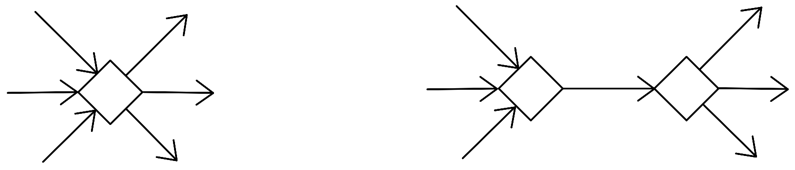


Figure 9: Merge/Decision Combination

5 FORK NODES

Fork nodes split flows into multiple concurrent flows. Control and data arriving at a fork are duplicated across the outgoing edges. No other behavior is associated with fork nodes in UML. For example, in Figure 10 control or data tokens leaving RECEIVE ORDER are copied by the fork, notated as a line segment, and passed to FILL ORDER and SEND INVOICE simultaneously. Since object tokens are only references to objects, copying them does not duplicate the objects themselves, only the references to them. There is no synchronization of the behaviors on concurrent flows in UML 2 activities, as there are in UML 1.x activities, which are a kind of state machines. In Figure 10, the flow to SHIP ORDER can complete long before SEND INVOICE is even finished, or vice versa[9,10].

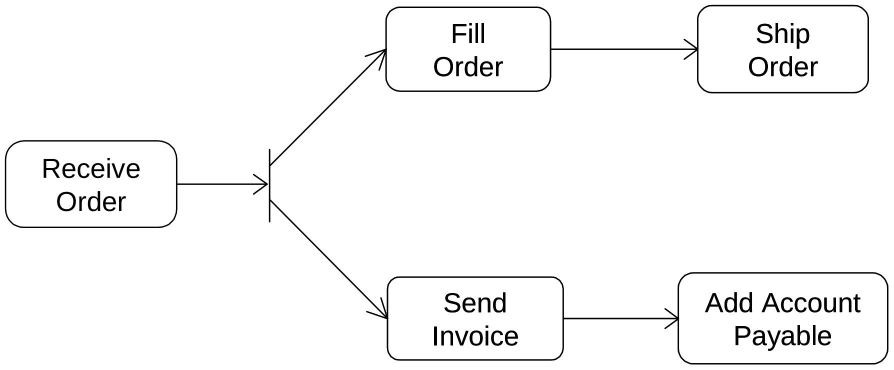


Figure 10: Fork Node

The default semantics for flows coming out of an action is that they are all initiated when the action completes. This creates concurrent flows, but data outputs from actions are not copied. The action outputs a separate value for each flow. Action outputs are also placed on pins, which are a kind of object node, and consequently hold values as they wait to move downstream. See the second article for more information on action outputs [2]. In UML 1.x, data flows are based on state transitions, so only one flow is initiated when the state (action) is exited [8]. See section 6 for analogous points about action inputs.

[9] Concurrent or orthogonal regions in state machines are synchronized through the run-to-completion semantics, which requires that behaviors invoked by the state machine complete before a new event is pulled from the input queue. This forces actions in concurrent regions to proceed in lockstep with each other. The "do" activity on states allows events to be processed while the activity is executing, but it also allows events to interrupt the do activity, which is not usually the desired effect in flow modeling.

[10] The current UML specification requires control and data tokens to either traverse all outgoing edges from a fork or none of them. This means if the outgoing edges have guards or other characteristics that prevent tokens from moving, that none of the concurrent flows will be initiated. The intention is for outgoing edges to start concurrent flows that are not otherwise prevented. This will be addressed in finalization.

6 JOIN NODES

Join nodes synchronize multiple flows. In the common case, control or data must be available on every incoming edge in order to be passed to the outgoing edge. Join nodes have the same notation as fork nodes, but joins have multiple edges coming in and one going out, whereas it is the opposite for fork nodes. Flows coming into a join are usually concurrent flows from an upstream fork. For example, Figure 11 shows a join node synchronizing two flows to CLOSE ORDER. Both SHIP ORDER and ADD ACCOUNT PAYABLE must complete before CLOSE ORDER can start.

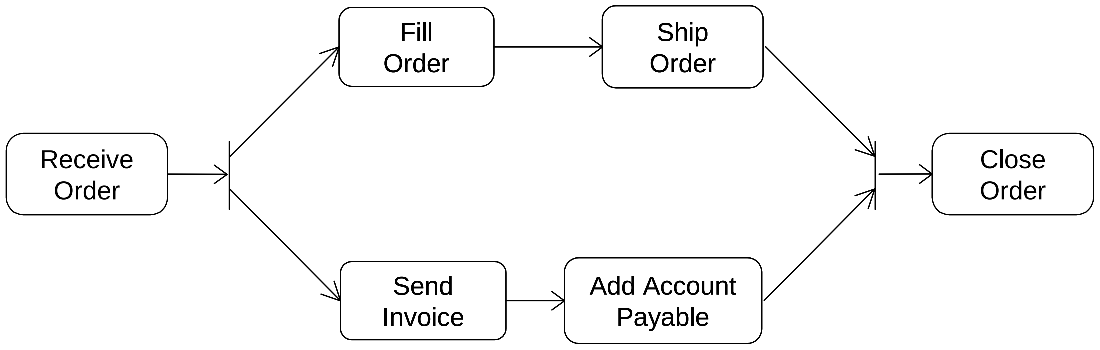


Figure 11: Join Node

Join nodes take one token from each of the incoming edges and combine them according to these rules:

1. If all the incoming tokens are control, then these are combined into a single control token for the outgoing edge.

2. If some of the incoming tokens are control and others are data, then these are combined to provide only the data tokens to the outgoing edge. The control tokens are destroyed.

For example, in Figure 11 the join combines control tokens from SHIP ORDER and ADD ACCOUNT PAYABLE into one, so that CLOSE ORDER is executed once instead of twice[11, 12].

[11] This requires one of the control tokens to be held somewhere while the other flow arrives, which is not technically possible, since control is output without pins. This will be addressed in finalization.
[12] It would be useful to have the option to combine object tokens for identical objects, especially in cases that two tokens are duplicate because they were copied by an upstream join.

The effect is the same if the join is omitted and the two flows go directly into CLOSE ORDER, because the action would wait for both of them anyway. It might be clearer to use the join, especially since UML 1.x activities would have needed only one flow to arrive, as with all state machines [8]. However, if the flows were carrying data, two tokens will be passed along the outgoing edge after synchronization, and go to a single pin in CLOSE ORDER. This would have the undesirable effect of CLOSE ORDER executing twice[13], and would not even be executable if the data is of incompatible types, because they would both be directed at the same input pin (see earlier articles for explanation of pins). For example, SHIP ORDER might output a tracking record, and ADD ACCOUNT PAYABLE the new account payable, both of which are needed as input to CLOSE ORDER. In this case, the data flows should be directed to two pins on CLOSE ORDER, without the join, as shown in Figure 12. This is another example of model refinement. Figure 11 might be taken as a process sketch and refined later into Figure 12, when it is clear what inputs are needed to close an order.

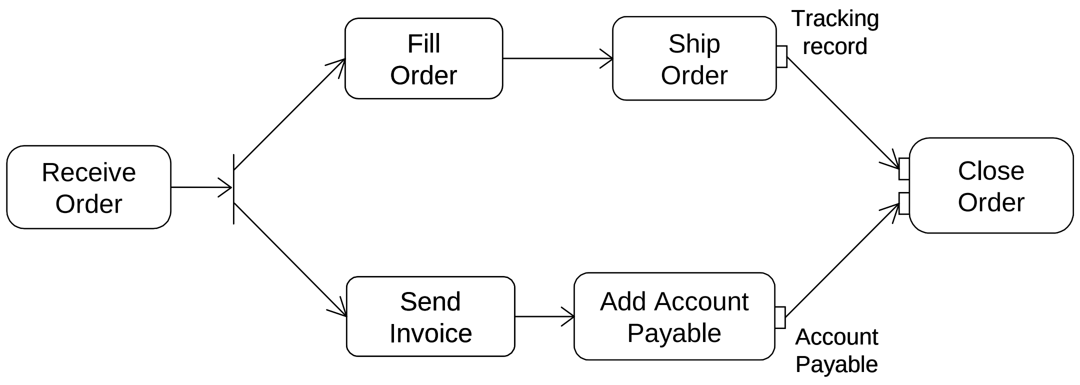


Figure 12: Joining Data Flows with Pins

Modelers should ensure that joins do not depend on control or data flows that may never arrive. For example, in Figure 13 when the problem report is not a high priority, the top flow is directed to a flow final (see next section), so control will never reach the join. This is corrected in Figure 14. See equivalent diagram in Figure 17[14].

[13] This actually depends on the multiplicity of the input parameter. If the input parameter multiplicity on CLOSE ORDER has a lower bound of two, it will consume both tokens coming from the join in one execution of the action. Multi-token flows are discussed later in the series.

[14] This is a good situation to use edge connectors, which are a notational technique for shortening the length of activity edge arrows, by breaking them up into a beginning and ending segment. See Figure 211 of the UML 2 specification [1].

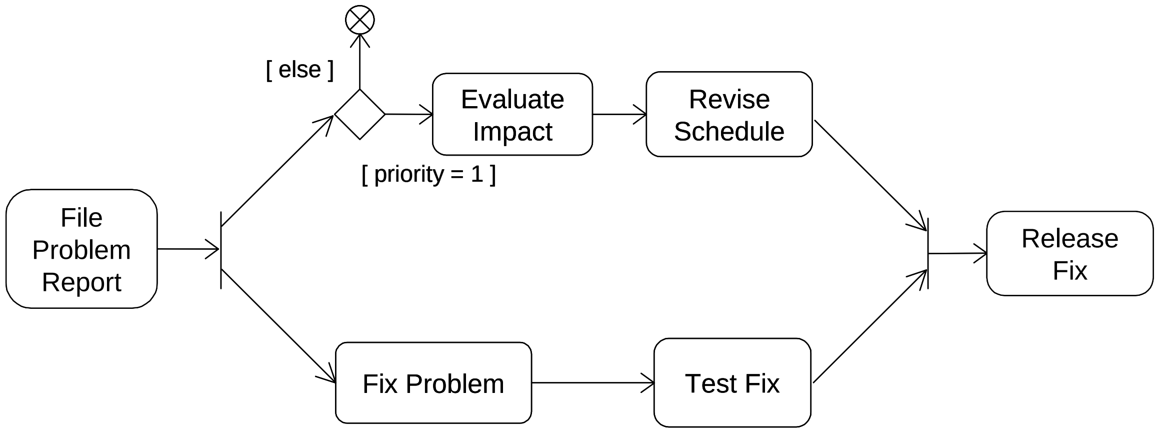


Figure 13: Join Node Anti-pattern

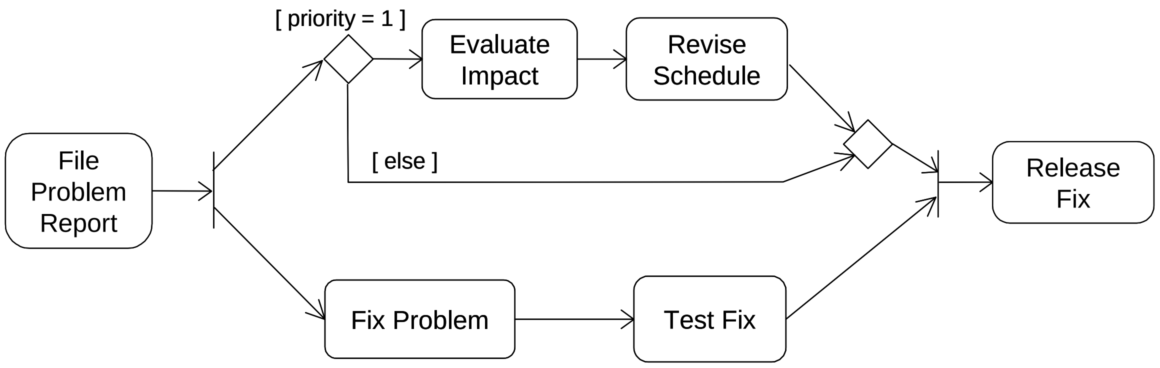


Figure 14: Join Node Pattern

It is not required that flows coming out of a fork be synchronized. For example, Figure 15 shows only some of the flows from a fork going to a join. The order is closed after it is shipped and invoiced, but the account payable might be monitored for a long period after that, so is not synchronized with closing the order. Concurrent flows can also be merged rather than joined, as shown in Figure 16. In this example, part inspection is serialized, while two parts can be made in parallel. The INSPECT PART action will be executed twice, once for each part arriving on concurrent flows[15]. This requires more than one token moving on the same flow line at one time. Multi-token flows are discussed later in the series. These are more examples of the expressiveness introduced in UML 2 activities over UML 1.x activities.

[15] It is also not required for flows coming into a join to be concurrent. For example, if a loop upstream generates alternate flows to a join, the synchronized flows will occur at completely different times.

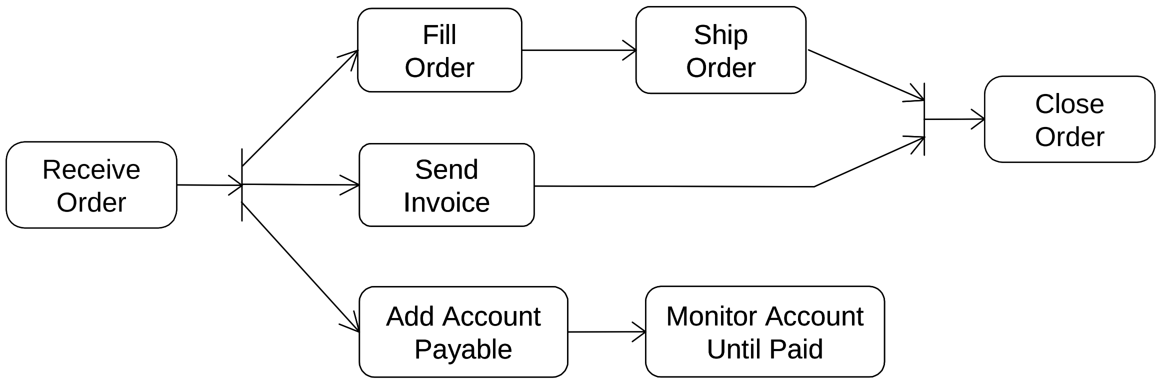


Figure 15: Fork with Partial Join

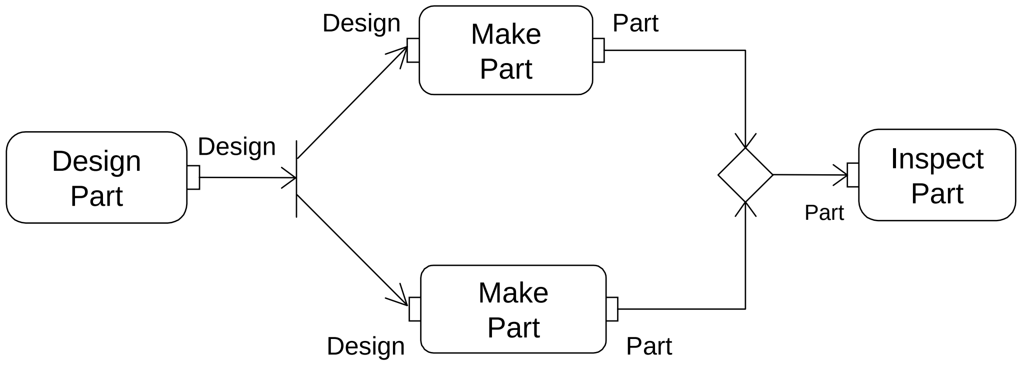


Figure 16: Fork with Merge

Modelers can specify the conditions under which a join accepts incoming control and data using a *join specification*, which is a Boolean value specification associated with join nodes. The default inherited from UML is "and", with the semantics described so far. Other join specifications can be given, using the name of the incoming edges to refer to the control or data arriving at the join. For example, Figure 17 shows an alternative to Figure 14 that substitutes a join specification for the merge node. The edges are named with single letters in this example, but can be any string.

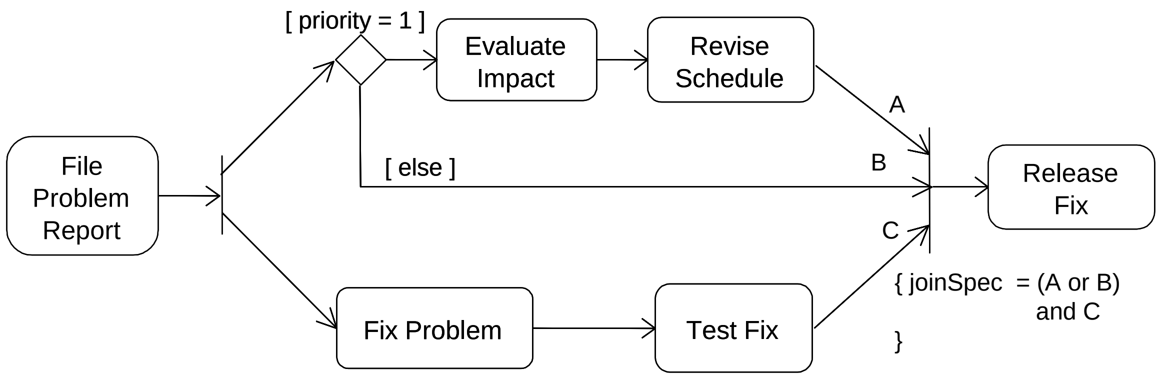


Figure 17: Join Specification

Figure 18 on the left shows a shorthand notation for a join immediately followed by a fork. It has the same effect as a separate join and fork, as shown on the right. Both have the same repository model, which contains separate join and fork nodes.

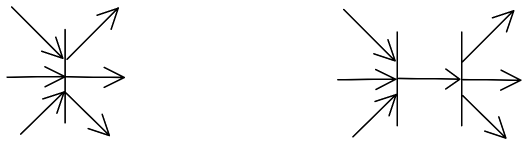


Figure 18: Join/Fork Combination

7 FINAL NODES

Flow in an activity ends at final nodes. The most innocuous form is the flow final, which takes any control or data that comes into it and does nothing. Flow final nodes cannot have outgoing edges so there is no downstream effect of tokens going into a flow final, which are simply destroyed. Since object tokens are just references to objects, destroying an object token does not destroy the object. Figure 19 extends Figure 10 with flow finals at the end. Each flow could have its own flow final and the effect would be the same. Activities terminate when all tokens in the graph are destroyed, so this one will terminate when both flows reach the flow final.

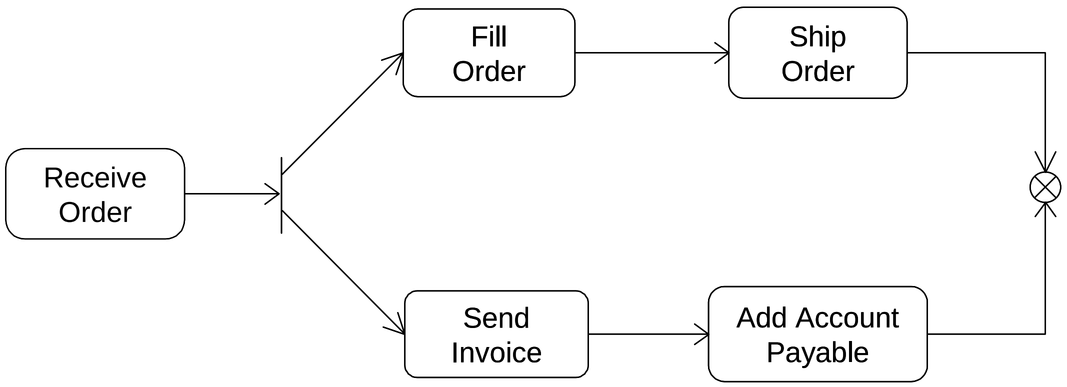


Figure 19: Flow Final Node

Activity final nodes are like flow final nodes, except that control or data arriving at them immediately terminates the entire activity. This makes a difference if more than one control or data token might be flowing in the graph at the time the activity final is reached, as in Figure 19. An activity final cannot be used instead of a flow final there because the completion of one concurrent flow would terminate the other[16]. In Figure 20 on the other hand, it does not matter whether a flow final or activity final is used, the execution traces are the same. Also each flow could have its own activity final on the end and the effect would be the same.

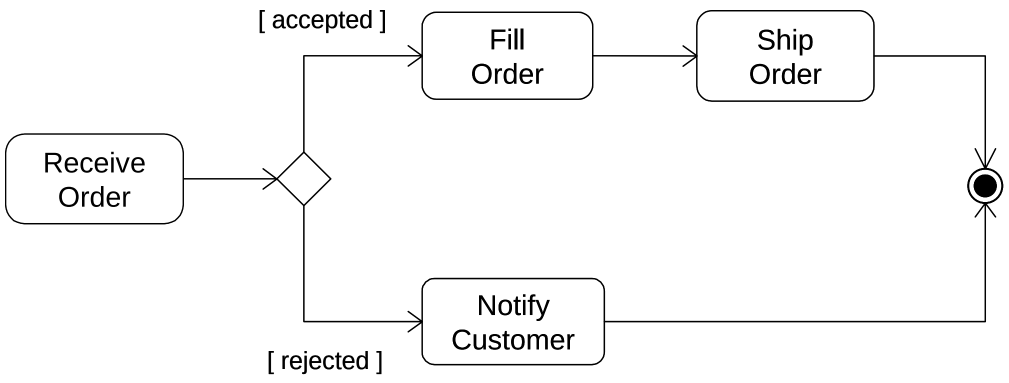


Figure 20: Activity Final Node

Figure 21 is an example where the termination functionality of activity finals is used in an intentional race between flows. This is a process for buying movie tickets by having people stand in separate lines until one gets the tickets for the group. The fastest line will produce a token to the activity final and terminate the other flow.

[16] This can be resolved by inserting a join after SHIP ORDER and ADD ACCOUNT PAYABLE that leads to an activity final. Then the activity would only terminate after both flows are done.

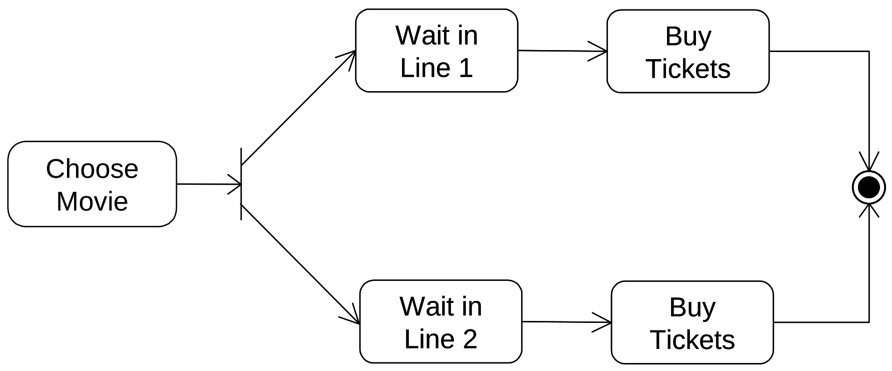


Figure 21: Activity Final Node, Racing Example

8 CONCLUSION

This is the third in a series on the UML 2 activity and action models. This article focuses on control nodes, which route control and data through an activity. The execution semantics of each kind of control node is described, along with the differences in concurrency from UML 1.x activities. UML 2 activities do not have the restrictions on concurrent flow that UML 1.x activities inherited from state machines. In particular, UML 2 concurrent flows are fully distributed in execution, not synchronized action-by-action as UML 1.x activities are. UML 2 forks and joins can be more flexibly paired with each other and other control nodes, rather than one-for-one as in UML 1.x activities. UML 2 action outputs and inputs also have concurrency and synchronization semantics, whereas they did not in UML 1.x.

ACKNOWLEDGEMENTS

Thanks to Evan Wallace and James Odell for their input to this article.

REFERENCES

[1] Object Management Group, “UML 2.0 Superstructure Specification,” http://www.omg.org/cgi-bin/doc?ptc/03-08-02, August 2003.

[2] Bock, C., “UML 2 Activity and Action Models, Part 2: Actions,” in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 41-56. http://www.jot.fm/issues/issue_2003_09/column4

[3] 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

[4] Object Management Group, “MOF 2.0 Query/Views/Transformations RFP,” http://www.omg.org/cgi-bin/doc?ad/02-04-10, April 2002.

[5] Shooter, S.B., Keirouz, W.T., Szykman, S., Fenves, S. J., "A Model for the Flow of Design Information in Product Development," *Journal of Engineering with Computers*, vol. 16, 2000, pp. 178-194.

[6] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[7] Bock, C., “Unified Behavior Models,” *Journal of Object-Oriented Programming*, vol. 12, no. 5, September 1999.

[8] Object Management Group, “OMG Unified Modeling Language, version 1.5,” http://www.omg.org/cgi-bin/doc?formal/03-03-01, March 2003.

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 at nist.gov.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 1, January-February 2004

UML 2 Activity and Action Models

Part 4: Object Nodes

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

This is the fourth 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 second two cover actions generally and control nodes [3][4]. The remainder of the series elaborates other specific elements. This article covers object nodes, which hold data and objects temporarily as they wait to move through an activity.

1 OBJECT NODES

To recap, UML 2 activities contain nodes connected by edges to form a complete flow graph. Control and data values flow along the edges and are operated on by the nodes, routed to other nodes, or stored temporarily. More specifically, action nodes operate on control and data they receive via edges of the graph, and provide control and data to other actions; control nodes route control and data through the graph; and object nodes hold data temporarily as it waits to move through the graph. Data and object are unified in UML under the notion of classifier, so the terms are used interchangeably. The term "token" is shorthand for control and data values that flow through an activity.

There are four kinds of object node, as shown in Figure 1 and described in the sections below. The functionality of object nodes is introduced in stages:

1) Holding a single token (section 2).
2) Holding multiple tokens, buffering, and backup (section 3).
3) Competing for tokens, traverse-to-completion semantics, deadlock prevention, and central buffers (section 4).
4) Data store nodes and a short history and future of data flow (section 5).

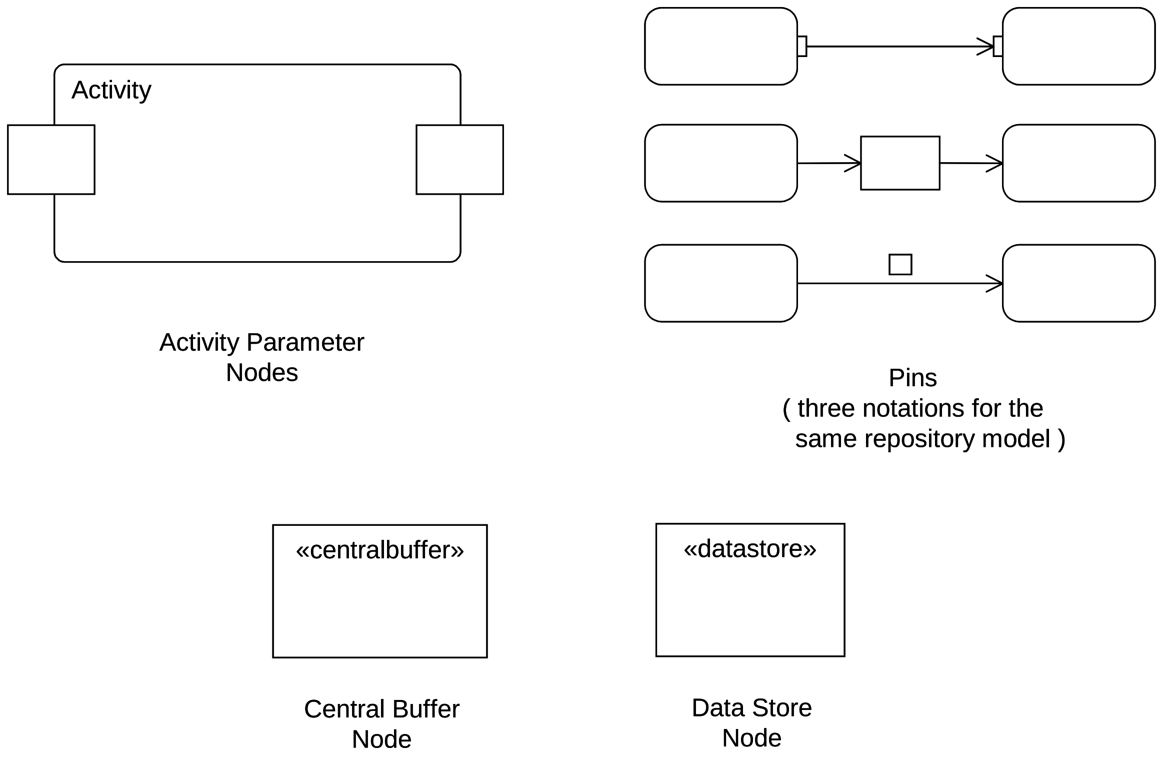


Figure 1: Object Nodes

2 PARAMETER NODES AND PINS

Previous articles introduced two kinds of object node: activity parameter nodes and pins. An example of parameter nodes is shown in the partial activity of Figure 2. It has two output parameter nodes on the right, each with a separate flow going into it. Whichever output value reaches a parameter node first is held there until the other arrives. When both parameter nodes have a value, the activity is complete and returns those values to the invoker of the activity.[1] The input parameter nodes on the left get their values all at once, when the activity is started. They may or may not be held there for some period, depending on whether they can flow downstream, as explained later in this article.

Parameter nodes must correspond to parameters of the containing activity. Activities are a kind of behavior in UML 2, and like all behaviors, they have parameters that specify the types of values that are input to the activity and output from the activity. Parameters on behaviors apply to all three kinds of behavior in UML, activities, state machines, and interactions, so are modeled separately from activity parameter nodes. See Figure 8 of the first article for a repository model showing the relation of behavior parameters and activity parameter nodes [2].

[1] The activity must also wait for all control and data to stop flowing in other parts of the graph before terminating.

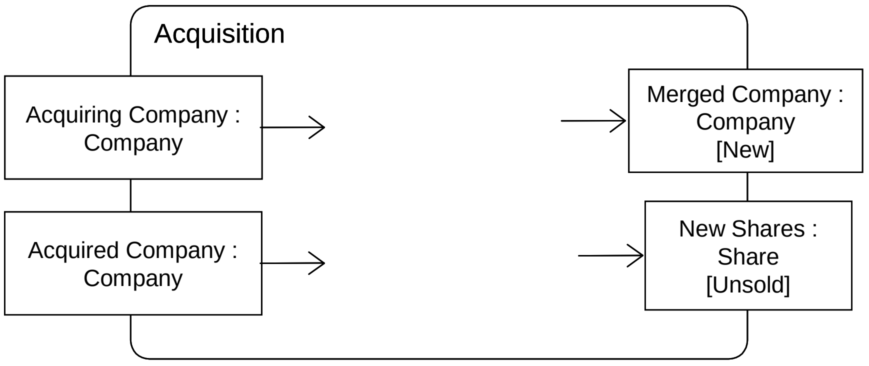

Acquisition
Acquiring Company : Company
Acquired Company : Company
Merged Company : Company [New]
New Shares : Share [Unsold]

Figure 2: Activity Parameter Nodes

All object nodes, including parameter nodes and pins, specify the type of value they can hold. In Figure 2, the parameter nodes hold values of type COMPANY and SHARE. If no type is specified, they can hold values of any type. Object nodes can also specify the state that their objects must be in, as provided by a state machine for the type of object being held. For example, in Figure 2 the objects held in the MERGED COMPANY parameter node must be in the NEW state and the objects in the NEW SHARES parameter node must be in the UNSOLD state. Objects must be in the required state before being put in the object node.[2] Multiple tokens in an object node can have the same value at the same time, for example, there can be multiple tokens for the number 3 or for the same instance of a class.

An example of input pins is shown in Figure 3, in two of the notational forms.[3, 4] Whichever input value reaches a pin first is held there until the other arrives. When both pins have a value, the values are passed into the action and it starts. If the action invokes the ACQUISITION activity in Figure 3, the input values move from the pins to the parameter nodes of the activity at the time of invocation. See section 4 of the second article for more about behavior invocation [3], and Figure 8 in particular, which shows the relation of pins to parameters.

[2] In this sense, the state requirement is an extension of the object node's type, which arguably should be promoted to types in general. Then they can be used by parameters, attributes, and other typed elements in UML. The same applies to constraints applied locally to a general type. This will be considered in finalization.

[3] All pin notations are stored in the repository the same way, which is analogous to the pin notation. See Figure 6 of the first article [2].

[4] A fourth notational form is defined for object nodes that have signals as their type. See Figure 275, page 350 of the UML 2 specification [1]. It will be discussed later in the series.

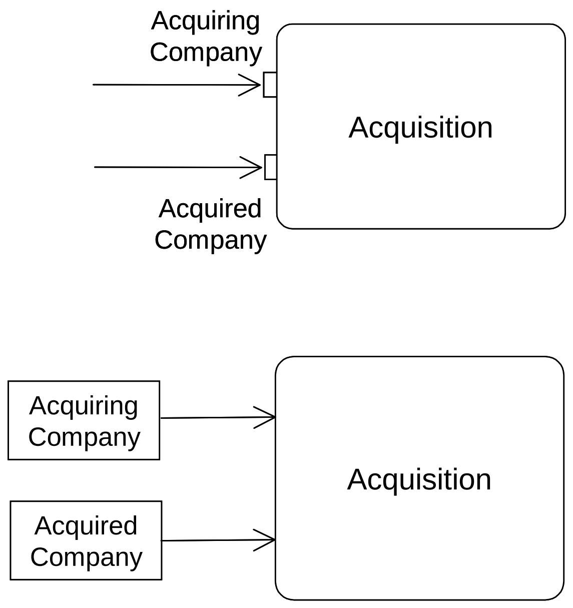


Figure 3: Input Pins

A special kind of input pin called a *value pin* is defined for providing constant values such as numbers, or values calculated by vendor or user-dependent expressions. It uses value specifications to model the value, described in the third article in connection with decision node guards [4]. It is notated like a normal input pin with a value specification written beside it. Unfortunately, value pins cannot be used to provide output values to activity parameter nodes. This will be addressed in finalization of UML 2.

Pins can be notated with the effect that their actions have on objects that move through the pin. Effect is one of the four values create, read, update, or delete. The example in Figure 4 indicates that Take Order creates an instance of Order and Fill Order reads it.5 The create effect is only possible on outputs, and the delete effect is only possible on inputs. If a single rectangle pin notation is used, then pin annotations such as effect still appear next to the action where the pin would have been shown.

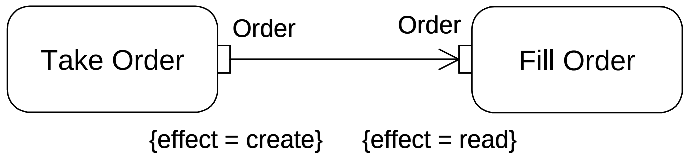


Figure 4: Effect

[5] The UML 2 specification inadvertently assigns effect to object flows, rather than to the parameters of behaviors. This will be addressed in finalization.

3 MULTIPLE TOKENS

Object nodes can hold more than one value at a time, and some of these values can be the same. Each object node specifies the maximum number of tokens it can hold, including any duplicate values, which is called the *upper bound*. At runtime, when the number of values in an object node reaches its upper bound, it cannot accept any more. Figure 5 shows an example using the buffering capabilities of pins between three manufacturing actions operating on parts. If painting is delayed too much for some reason, the input pin will reach its upper bound, and parts from polishing will not be able to move downstream. If painting is delayed further, the output pin of polishing will fill up and the polishing behavior will not be able to transfer out polished parts. Unless the polishing behavior has an object node internal to it that buffers output parts, it will not be able to take parts from its input pin, which will likewise fill up and propagate the backup. Only when the input pin to PAINT goes below its upper bound will parts be able to flow again.

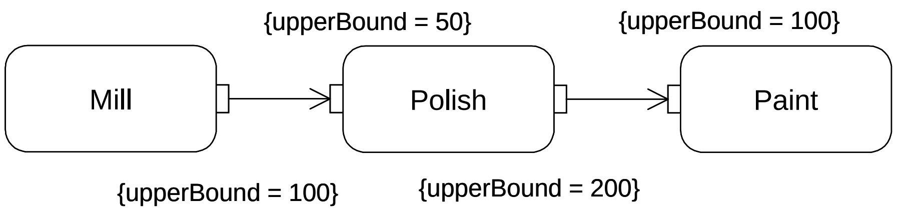


Figure 5: Upper Bound

Buffering capabilities are intentionally assigned to object nodes in activities, rather than to parameters of behaviors. The parameters of a behavior or operation, also known as the *signature*, only declare the kinds of things needed for input and output, and how many of each. Buffering capabilities are assigned either to pins on actions that invoke behaviors, or to the implementations of a behavior, such as parameter nodes in activities. The UML 2 metamodel separates pins and activity parameter nodes from parameters of behavior. See example repository model in Figure 8 of the first article of the series [2].

Some applications have the advantage of executing the same behavior concurrently to reduce backup restrictions. For example, a factory executing the process in Figure 5 might have more than one station to use for the PAINT step. This means that more than one part arriving at the input pin of PAINT can start an invocation of PAINT at the same time, or at staggered times. Likewise, the concurrent executions of PAINT can put more than one part on the output pin at the same time, or at staggered times, and not necessarily in the same order in which they were taken from the input pin. UML 2 calls this a REENTRANT behavior.[6,7,8] It is indicated with the keyword «reentrant» on the action, or with a property list {reentrant}.

[6] The term "reentrant" in computer science means a procedure that can have multiple executions occurring at the same time without interfering with each other. This applies to UML 2 reentrant behaviors, but the term is extended to have the particular execution semantics for activities described above.

Software applications can especially make use of reentrancy, for example, when processing packets from a telephone switch to provide information for a billing system. Multiple threads can be set up for each step in the processing so packets taking a short time to handle do not need to wait for those taking longer. Upper bounds can be set very high to temporarily buffer packets or intermediate results if the billing system goes down.

Object nodes holding multiple values can specify the order in which values move downstream. The default is first-in, first-out (FIFO, a pipe), but users can change this to last-in, first-out (LIFO, a queue), or specify their own behavior to select which value is passed out first. For example, Figure 6 shows orders being filled using a priority ranking. The user-specified selection behavior is passed all the values in the object node and returns one to move downstream.[9,10,11] Selection behaviors can also be used on object flow edges coming out of object nodes. This is useful in situations where the selection criteria varies with the path taken out of the object node, see sections 4 and 5.

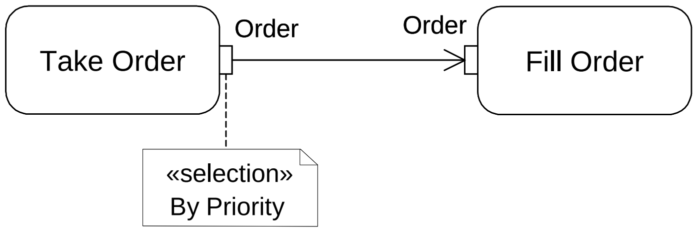


Figure 6: Selection Behavior

A partial repository model for Figure 6 is shown in Figure 7. The selection behavior accepts multiple orders from the object node and returns one that should be offered next to an outgoing edge. Parameter multiplicities are described next and shown in Figure 7 as the LOWERVALUE and UPPERVALUE of parameters.

[7] UML 2 does not restrict the number of concurrent executions of a reentrant behavior that can exist at one time. This will be addressed in finalization or a profile.

[8] Reentrant behaviors cannot have streaming parameters [3], because it would be not be possible to determine which execution of the behavior should receive a streaming value at runtime.

[9] It currently is not specified what order values in an object node are passed to selection behaviors. It would be most useful to pass them in the order they arrived at the object node, so the selection behavior can, for example, use FIFO within order priority, or reverse it for LIFO. This will be addressed in finalization.

[10] This way of ordering values has the benefit of dynamically responding to current conditions, because the selection behavior can account for these conditions every time a value is chosen to move downstream. However, for applications in which the selection criteria are fixed, it is more efficient to insert new values into the object node according to the ranking. Then selecting a value to move downstream is just a matter of taking the one at the top of the list, rather than reevaluating the order each time. A selection behavior can be implemented as an insertion-time ordering, because the activity model only specifies the required runtime effect. However, it is difficult for tool vendors to automatically translate a selection algorithm into a queue insertion algorithm. This will be addressed in finalization.

[11] The selection behavior could alternatively be on the input pin of FILLORDER. The action will consume values in the order of selection.

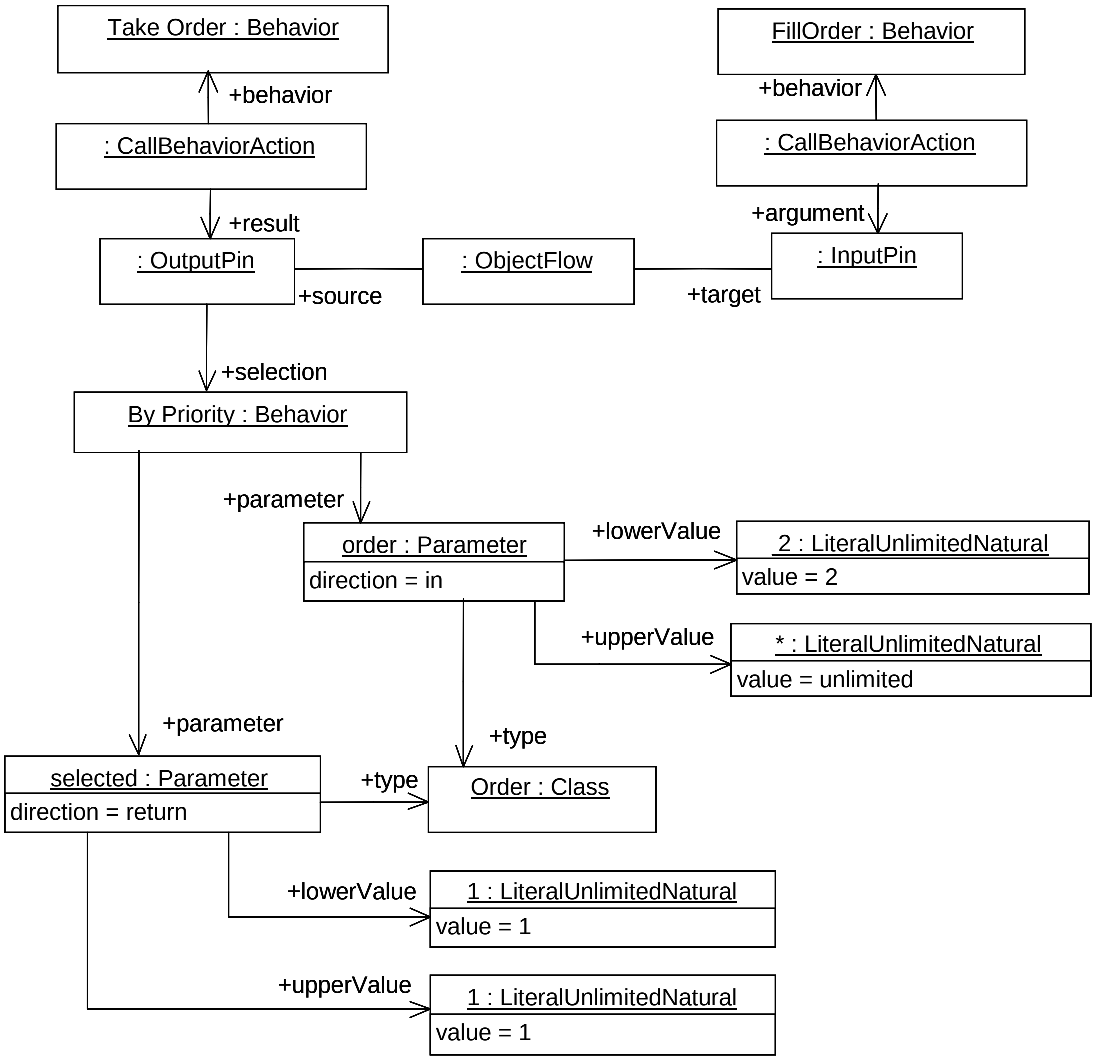


Figure 7: Repository for Figure 6

Behavior and operation parameters can have multiplicities that specify the minimum and maximum number of values each parameter accepts or provides at each invocation of the behavior. Minimum multiplicity on an input parameter means a behavior or operation cannot be invoked by an action until the number of values available at each of its input pins reaches the minimum for the corresponding parameter, which might be zero (see the second article on actions that invoke behaviors [3]).[12] For example, Figure 8 shows an action invoking a behavior for playing baseball, which might be delayed waiting for all nine the players to arrive. The pin label reflects the information in the PLAYER parameter

[12] An action invoking a behavior with one input parameter of zero minimum multiplicity could in theory begin executing spontaneously and repeatedly because it has all the data inputs it requires. However, the only reasonable interpretation here is that the action needs either a data input or a control input to start. This will be clarified in finalization.

of the PLAY BALL! behavior, including its type and multiplicity. On the other hand, if play is delayed waiting for the equipment, and meanwhile more than nine people collect at the input pin, then only the maximum number, nine, are taken to start the action.[13,14,15,16]

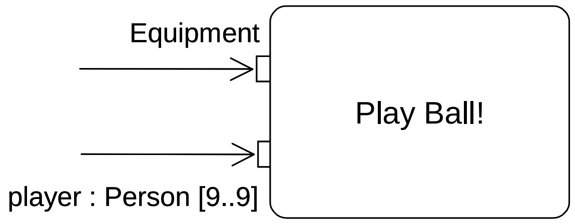


Figure 8: Parameter Multiplicity

Similar to minimum multiplicity on parameters is weight on object flow edges, which specifies the minimum number of values that can traverse an object flow edge at one time. For example, in Figure 9 the MAKE PART action must output 100 parts before they can move to the input of SHIP PART. SHIP PART can take from 1 to 1000 parts, because it is a general purpose behavior, but in this particular usage of it, parts are shipped in batches of 100. If for some reason shipping is delayed, multiples of 100 parts will collect at the input of SHIP PART, whereupon more than 100 parts will be shipped at the next invocation, unless there is an upper bound on the input pin. A weight of "all" means that all values in the source object node are moved at once. The default weight is 1.

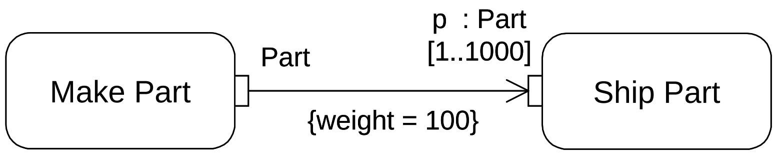


Figure 9: Object Flow Weight

[13] UML does not define a default parameter multiplicity to apply to the EQUIPMENT input above, but modelers would probably expect it to be exactly one. This will be addressed in finalization.

[14] Parameters with maximum multiplicity greater than one can be marked as ordered. This means that each invocation of a behavior using the parameter can input or output multiple, ordered, runtime values. For pins corresponding to those parameters, it is not currently specified that the values in an input pin will be passed in the same order to the parameter on invocation, or from parameter to output pin on termination. This is a reasonable expectation, however, and will be addressed in finalization.

[15] Parameters with maximum multiplicity greater than one can be marked as allowing multiple occurrences of the same value. This means that a single invocation of a behavior using the parameter can input or output multiple runtime values where some of the values are the same. Since object nodes with upper bound greater than one always allow multiple tokens to have the same value, it is a good idea to make this indication on parameters also, unless the modeler knows in advance that it will never happen.

[16] The interaction of multiplicity and streaming [3] is not currently specified. One option is that the minimum and maximum give restrictions on size of "batches" that can stream in or out at one time.

4 TOKEN COMPETITION

A parameter node or pin may have multiple edges coming out of it, whereupon there will be competition for its tokens, because object nodes cannot duplicate tokens like forks can [4]. Modelers should use this pattern only if they want indeterminacy in the movement of data in the graph. For example, Figure 10 shows parts being made, then painted at one of two stations, but not both.

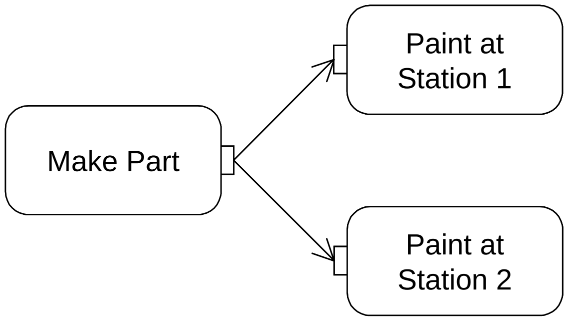


Figure 10: Token Competition

Figure 10 is also an example of how edges cannot hold tokens, as object nodes and actions can. If the input pin of PAINT AT STATION 1 is full, the object flow edge going into it cannot claim a value from the output of MAKE PART and hold it until PAINT AT STATION 1 is able to take it. The token remains at the output of MAKE PART until the traversal can be completed to one of the input pins. The terminology of the UML 2 specification is that the output pin "offers" the token to the outgoing edges, which in turn offer it to their respective targets. The traversal of the edge cannot take effect until all the elements between source and destination object node accept the offer, including the destination. This article calls the principle *traverse-to-completion*.

Control nodes cannot hold tokens, either. For example, Figure 11 shows a decision node routing some parts for testing and others for painting (see the previous article on decision nodes and guards [4]). If a part output from MAKE PART fails the testing guard and the input pin at PAINT is full, then the part cannot reside at the decision node waiting to be painted. It remains at the output pin and will be routed either to testing when that guard succeeds or to painting when the input pin of PAINT is no longer full. If multiple edges were coming out of the output pin of MAKE PART, then the part would be subject to competition, and may not ever be painted or tested at all.[17]

[17] Figure 11 would have the same effect if the decision node were removed, and the edges with guards came directly from the output pin of MAKE PART. The purpose of decision nodes is to ensure values move along exactly one of its outgoing edges, which is also the semantics of object nodes. The ELSE guard is currently specified only for edges coming from decision nodes, but is equally applicable to edges from object nodes. This will be addressed in finalization.

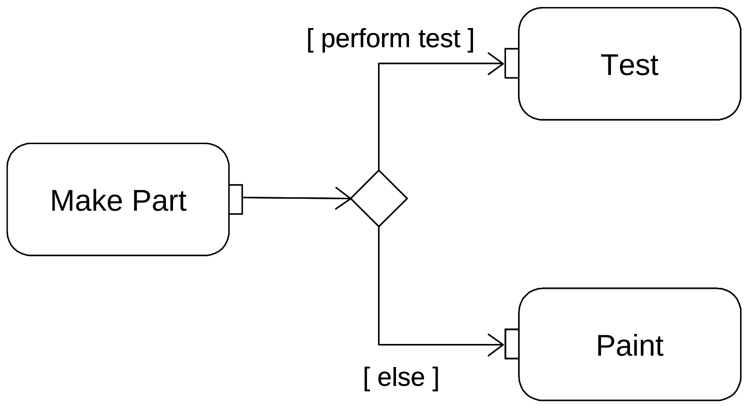


Figure 11: Decision Node

Preventing control nodes and edges from holding tokens ensures that values do not get "stuck" when alternative paths are open. In any particular direction of flow it may take a long time to select tokens, decide how to route them, for backups to clear, and so on. Traverse-to-completion means that tokens move along the path of least resistance by going to the first available object node. Data and object values are always residing in object nodes or being operated on by actions, moving instantly between them when all the criteria along the path between source and destination are satisfied. The decision of where to route tokens may take time, but no tokens move until the decision process is complete. For this reason, behaviors associated with traversal, such as decision input and selection should not have side-effects or be overly complicated, because they might be executed many times before a value succeeds in being moved.[18,19]

Another behavior that falls under traverse-to-completion is the transformation of tokens as they move across an object flow edge. Figure 12 shows customers being retrieved from orders. Each order is passed to the transformation behavior and replaced with the result. The result is offered to the input pin of SEND NOTICE and must be accepted there before the order can be removed from the output pin of CLOSE ORDER. The repository stores a complete behavior, but the notation can just show the contents or an abbreviation of the behavior, as in Figure 12.

[18] Traverse-to-completion enables implementations to optimize the execution of traversal behaviors. For example, if the destination object node is full, a selection behavior at the source object node need not be executed until the destination is ready to accept values.

[19] The division of activity nodes in this manner is similar to the distinction between states and pseudo states in UML state machines. A state machine cannot pull events from its event buffer while it is transitioning between states, including while in pseudostates. Informally speaking, a UML state machine can only "rest" in real states, which is called the *run-to-completion* requirement. The purpose is to handle each event completely before taking another event. This ensures every state completes its actions without interruption from incoming events.

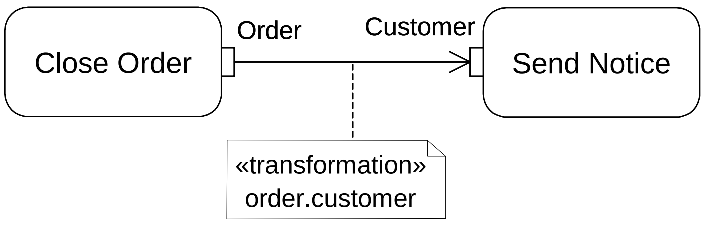


Figure 12: Object Flow Transformation

Central buffers are for situations where tokens under competition arrive from multiple sources. For example, Figure 13 shows parts arriving at a central buffer from two factories, which are then painted at two other factories. Pins can be omitted from the notation, but they are still recorded in the repository. Pins cannot be used as central buffers, because pins have flows coming or going out, but not both.

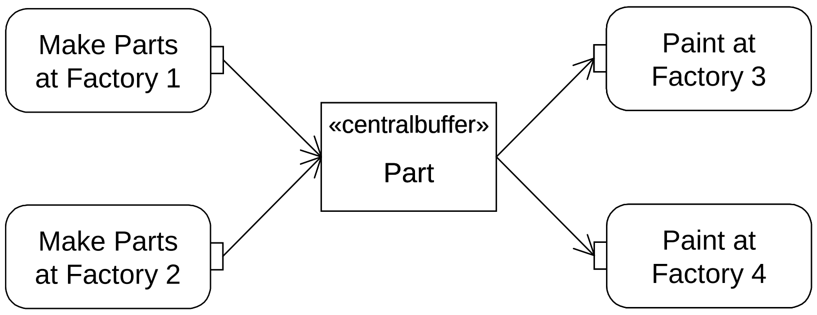


Figure 13: Central Buffer

Another aspect of traverse-to-completion is that an input pin of an action cannot accept tokens until all the input pins of the action can accept them. This is to prevent deadlock, where the input pins of two actions each have some of the tokens required for the other to start. For example, Figure 14 shows two drilling behaviors requiring a drill and an extension cord to start, as might happen when two carpenters are working together (pins for material being drilled are omitted). One action's input pin cannot accept the drill when the other input pin on that action cannot get the extension cord at the same time. This prevents one action from holding the drill while the other takes the extension cord, and neither can start. The example is adapted from the well-known dining philosopher's problem in models of concurrency [5]. Input pins can still buffer up multiple tokens, but they can only accept tokens in unison with the other input pins on the same action.[20]

[20] Input pins of invocation actions must take enough tokens to meet the minimum multiplicity of the corresponding parameter.

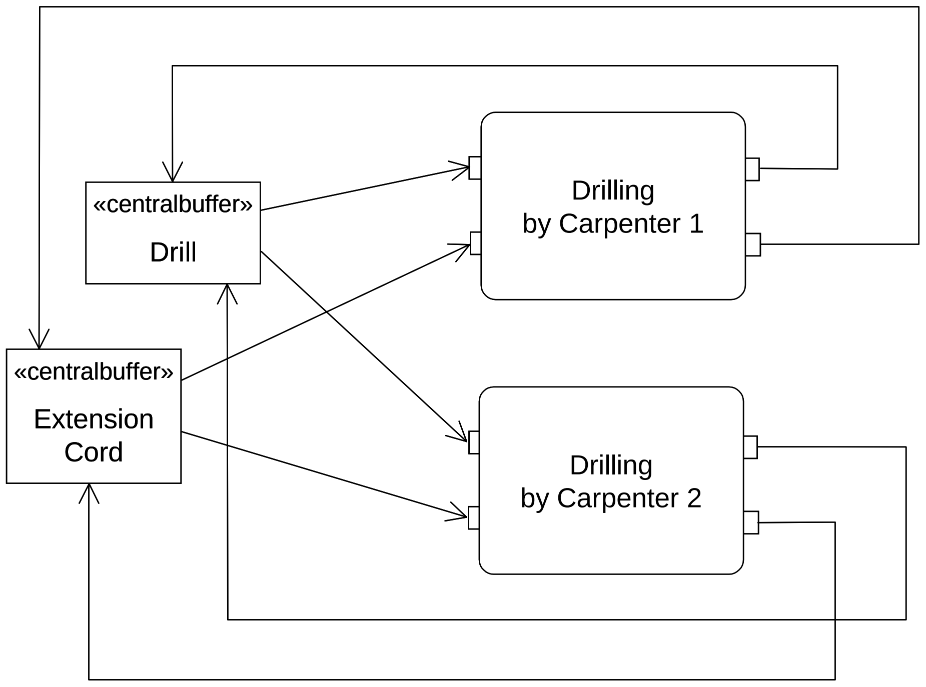


Figure 14: Avoiding Deadlock

5 DATA STORE NODES

Earlier forms of data flow and storage have the following characteristics [6][7]:

- *Passive*: the presence of data in the store does not initiate actions. Actions take data as needed.
- *Non-depleting*: the use of data in the store does not remove it from the store.
- *Persistent*: data in the store remains there after the activity containing it terminates.

This might be informally called the "pull" form of data flow and storage. Later forms of data flow and storage, including UML 2 object nodes, have exactly the opposite characteristics, which might be called "push":

- *Active*: the presence of values in an object node initiates downstream actions by sending inputs to them.
- *Depleting*: values in an object node used by an outgoing edge are not available to other outgoing edges. This is token competition.
- *Transient*: values do not remain in object nodes after the activity containing the object node terminates.[21]

[21] Tokens are only references to objects, so the objects themselves are not deleted, even if the activity is terminated, for example with an activity final [4].

UML 2 data store nodes are an attempt to support the earlier form of data flow and storage by providing a non-depleting specialization of object node. Tokens flowing out of data store nodes are copies of tokens that remain in the data store node, so the values seem as if they are being read from the store. Tokens in a data store node cannot be removed, though values do not remain in the store after the containing activity is terminated, and a token arriving at a store that already has another token for the same object replaces that token.

Selection and transformation behaviors can be applied on edges coming out of data store nodes to retrieve information from the store, as if a query were being performed. For example, a selection behavior can identify an object to retrieve and the transformation behavior can get the value of an attribute on that object. Figure 15 shows a personnel data store being populated by a HIRE EMPLOYEE behavior, and read by other behaviors. Employees not assigned to projects are selected for input to the ASSIGN EMPLOYEE behavior. Once a year, all employees are reviewed, using the ACCEPTEVENTACTION as a timer, described later in the series. This pattern uses traverse-to-completion to ensure that the employee list is read only once a year, since the stored objects are only retrieved when the join succeeds.

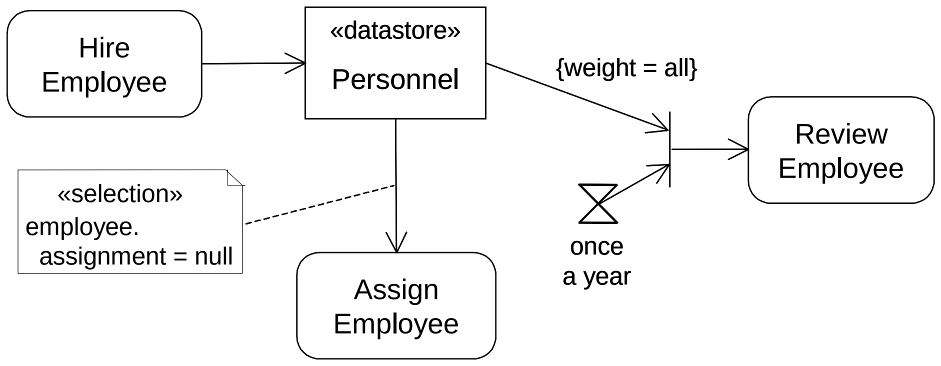


Figure 15: Data Store Node

UML data store nodes are still active and transient, however, and do not completely capture pull semantics.[22] Since the earlier and later forms of data flow and storage are so different, the most accurate way to model the earlier forms in the later is to use actions instead of flows. The functionality of earlier data storage can be achieved with UML 2 actions for modifying persistent objects, such as ADDSTRUCTUREFEATUREVALUEACTION, described later in the series. A data flow going into an earlier form of data store is equivalent to assigning that data as a value of an attribute of a persistent object.

[22] Implementation-dependent extensions can support selection behaviors that succeed only when a downstream action has control passed to it, partially capturing pull semantics. This still does not model the fact that earlier forms of data store could be read anytime during an action, not just at start-up of actions. In UML this would require extensions to streaming [3].

Conversely, a data flow coming out from a traditional data store is equivalent to retrieving a value from an attribute in a persistent object using UML 2 actions for that. One could imagine extending UML with a concise graphical or textual notation for these patterns of using read and write actions on persistent objects.[23]

The transition from earlier to later forms of data flow and storage indicates a trend of unifying control and data flow. The characteristics of later data flow are more like control (active, depleting, transient). Conversely, recent work in UML for Systems Engineering treats control as a form of data by providing additional control values for terminating actions, as well as control queuing, and control operators [8][9]. The trend in unification of control and data functionality forms a cycle, because new capabilities for one suggest new capabilities for the other. For example, if control has terminating values, why should data be limited to being a form of enabling control? If data arrives at an action that is already executing, it might mean the old data is incorrect, and the action should be terminated, and start over. Further unification of control and data is an area for future work.

6 CONCLUSION

This is the fourth in a series on the UML 2 activity and action models. It focuses on object nodes, which hold data and objects as they wait to move through a flow graph. The four kinds of object node are covered: parameter nodes, pins, central buffers, and data stores. Functionality is addressed in stages, starting with single and multiple token flow, then token competition, traverse-to-completion semantics, and deadlock prevention. The article ends with a short history and future of data flow models.

ACKNOWLEDGEMENTS

Thanks to Evan Wallace and James Odell for their input to this article.

[23] A hybrid approach would be to extend data store nodes with the capability of modifying persistent objects. For example, a data store node could be assigned an object and attribute in which to store values received by the node. The object could be represented by a partition. Partitions will be discussed later in the series. It is being considered in a submission to UML for Systems Engineering [8].

REFERENCES

[1] Object Management Group, "UML 2.0 Superstructure Specification," http://www.omg.org/cgi-bin/doc?ptc/03-08-02, August 2003.

[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.

[3] Bock, C., "UML 2 Activity and Action Models, Part 2: Actions," in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 41-56, http://www.jot.fm/issues/issue_2003_09/column4

[4] Bock, C., "UML 2 Activity and Action Models, Part 3: Control Nodes," in *Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 7-23, http://www.jot.fm/issues/issue_2003_11/column1.

[5] Filman, R., Friedman, D., *Coordinated Computing: Tools and Techniques for Distributed Computing*, McGraw-Hill, 1984.

[6] Rumbaugh, J., et al., *Object-oriented Modeling and Design*, Prentice Hall, 1991.

[7] Shlaer, S., Mellor S., *Object-oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

[8] OMG Systems Engineering DSIG, "UML for Systems Engineering RFP," http://www.omg.org/cgi-bin/doc?ad/03-03-41, March 2003.

[9] Bock, C., "UML 2 Activity Model Support for Systems Engineering Functional Flow Diagrams," Journal of the International Council on Systems Engineering, vol. 6, no. 4, October 2003.

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 at nist.gov.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 7, July-August 2004

UML 2 Activity and Action Models

Part 5: Partitions

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

This is the fifth 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 three cover actions generally, control nodes, and object nodes. This one describes partitions, which are a way of grouping actions that have some characteristic in common. In particular, they can relate actions to classes that are responsible for them, and highlight the abstraction that activities provide for interaction diagrams and state machines.

1 PARTITIONS

Partitions are groups of actions that highlight information already in an activity, or that will be, and present it in a more compact way. Partitions do not have execution semantics themselves, but because they are redundant with the information in the executable part of the model, tools can automatically update the executable model when the user modifies partitions or their contents. To reduce clutter, tools can also omit the redundant portions of the execution from the diagram, while still keeping them in the model repository for system generation. Figure 1 shows the example used in this article, adapted from [1][3].[1] Each of the areas between the parallel, vertical lines is a partition, and this particular way of notating them is called a *swimlane*. An alternate notation is shown in Figure 2, where partitions are labeled on nodes.

[1] Forks and join are shown for clarity, as suggested by [3], but are not necessary in this example, due to similar semantics for actions [4].

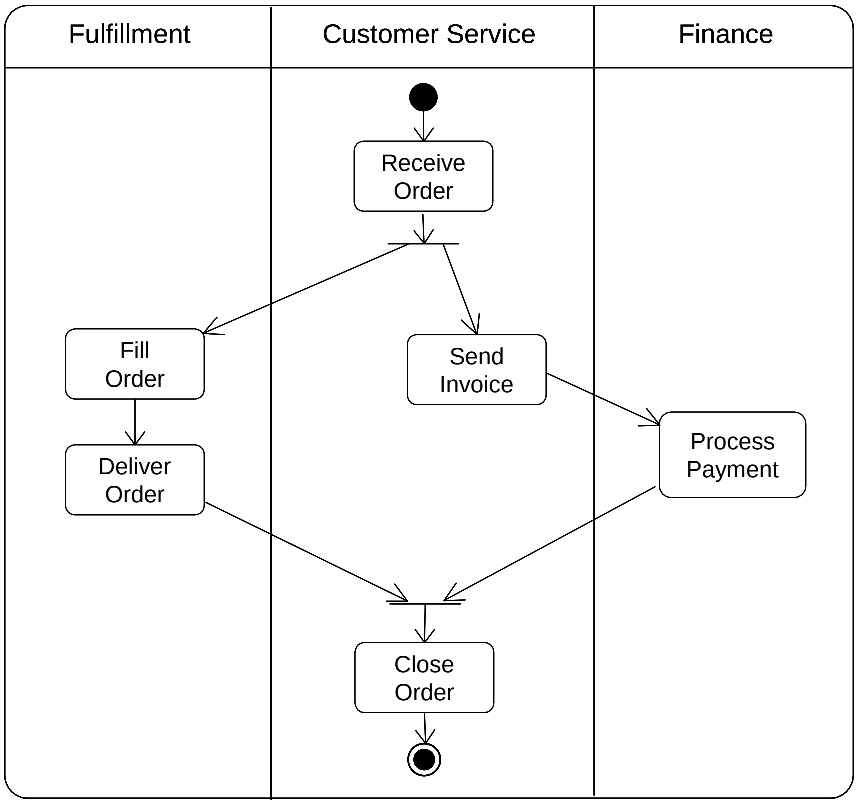


Figure 1: Partition Example, Swimlane Notation

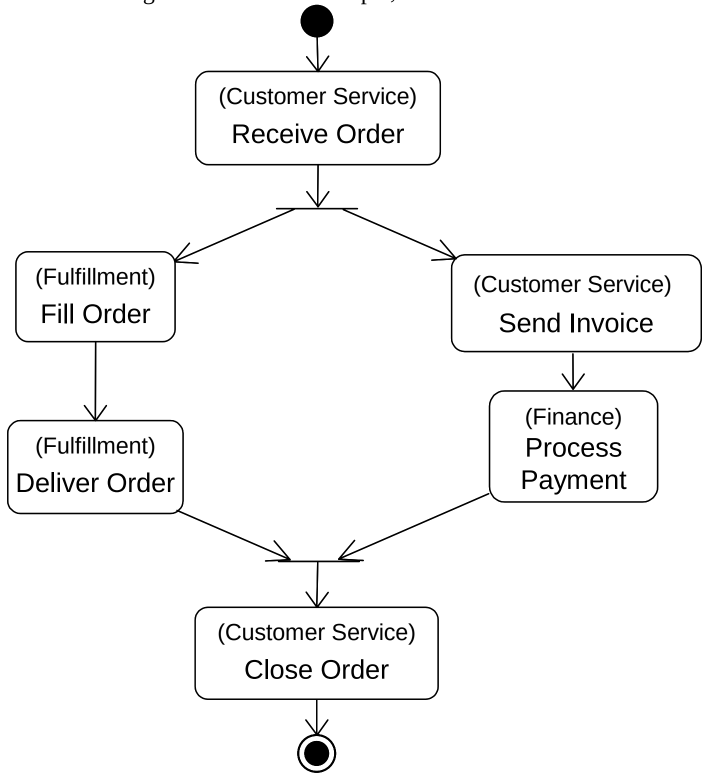

(Customer Service) Receive Order
(Fulfillment) Fill Order
(Customer Service) Send Invoice
(Fulfillment) Deliver Order
(Finance) Process Payment
(Customer Service) Close Order

Figure 2: Partition Example, Node-based Notation

Because there is so much information in an activity, and so many ways to highlight it and make it more compact, partitions can be extended by modelers and tools to support applications not explicitly defined in UML. This article describes the ways of using partitions predefined in UML and gives an example of a modeler-defined application. It also discusses the translation of activities to interaction diagrams and state machines in sections 3 and 6.

2 CLASS PARTITIONS

Partitions are often used to indicate what or who is responsible for actions grouped by the partition. The term "responsible" has a wide variety of meanings, but the one defined by UML is that a class supports the behavior invoked by actions in the partition. For CALLOPERATIONACTION, this means the class defines the invoked operation [4]. For CALLBEHAVIORACTION, it means that the class owns the behavior.[2] For example, Figure 3 shows partitions representing classes, as would be appropriate to model generic systems that operate in whatever company installs them.[3] If the action FILL ORDER is to invoke an operation, then the operation must be declared on the FULFILLMENT class. This is shown on action PROCESS PAYMENT with the notation for CALLOPERATIONACTION that indicates the target class of the invocation [4]. When all the CALLOPERATIONACTIONS conform to their containing partitions, the partitions only highlight information already in the activity. The companion class model is shown in Figure 4.[4] Each company will have instances of these classes, which are targets of the operation calls. See second half of this section, and Figure 7.

[2] Same applies to behaviors on nodes other than actions, for example, decision input behaviors on decision nodes [5].

[3] The guillemet notation is used for metaclasses as well as stereotypes, which are both metalevel concepts. Keywords refer specifically to an aspect of the UML metamodel, such as metaclasses or metaproperties.

[4] An alternative approach use partitions representing order and invoice classes, with operations on them instead. This conforms to the conventional object-oriented development style of translating objective nouns to classes and verbs to operations on them. Figure 3 is more typical of web services or agent techniques, which identify active entities that operate on passive ones. These approaches highlight the responsible parties, but have the disadvantage of being more brittle under organizational or industrial change [6]. The alternatives could be harmonized by class partitions representing the sender of the message/operation, combined with other dimensions representing the targets (dimensions are covered in section 5). This will be addressed in revision.

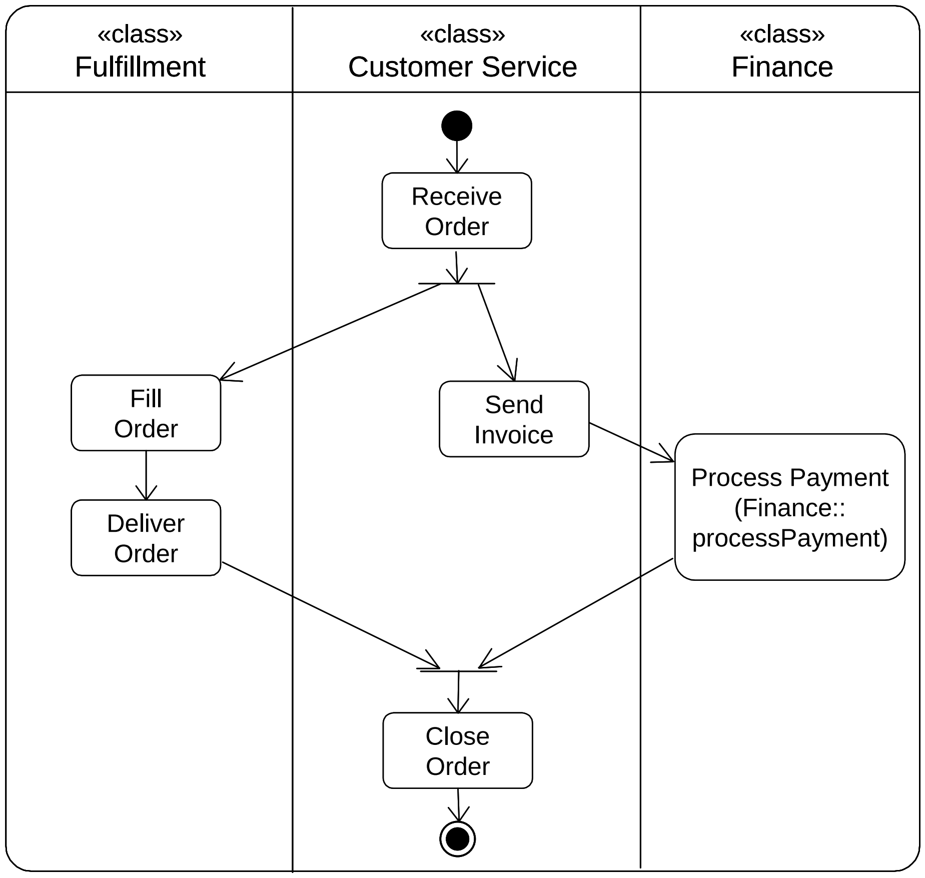


Figure 3: Partitions Representing Classes

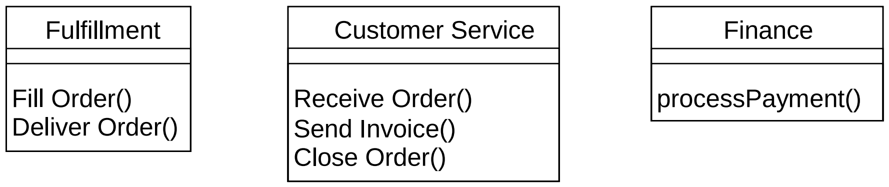


Figure 4: Class Model for Figure 3

Tools can add value to class partitions by automatically maintaining consistency between partitions, actions, and classes as the diagrams are modified. For example, when a CALLOPERATIONACTION action is moved into the FULFILLMENT partition, the tool can automatically move the invoked operation to the corresponding class, from whatever class it was on previously. For applications that are strictly object-oriented, if a class partition is added, the CALLBEHAVIORACTIONS in it can be converted to CALLOPERATIONACTIONS, and operations automatically defined on the class with the behaviors used as a methods. Tools can also add value by making the diagrams more compact while keeping the full specification in the underlying model repository. For example, the diagram can omit the full CALLOPERATIONACTION notation, which is redundant with the partitions, while leaving the action completely specified in the underlying model.[5]

Once partitions, actions, and classes are consistent, there is the question of which particular instances are targets for the CALLOPERATIONACTIONS. There are a number of ways to show this. One is to add object flows that provide the instance as input to CALLOPERATIONACTION, as shown in Figure 5. This has the obvious disadvantage of clutter, but is explicit about the necessary inputs to the actions. It also makes clear that the customer service department sending the invoice should be the same one that closes the order. An alternative is to use partitions that represent instances directly, but this is only useful for individual scenarios, not for specifying a behavior that must operate on many instances. And it would still require object flows from value pins to pass instances to the CALLOPERATIONACTION [4].

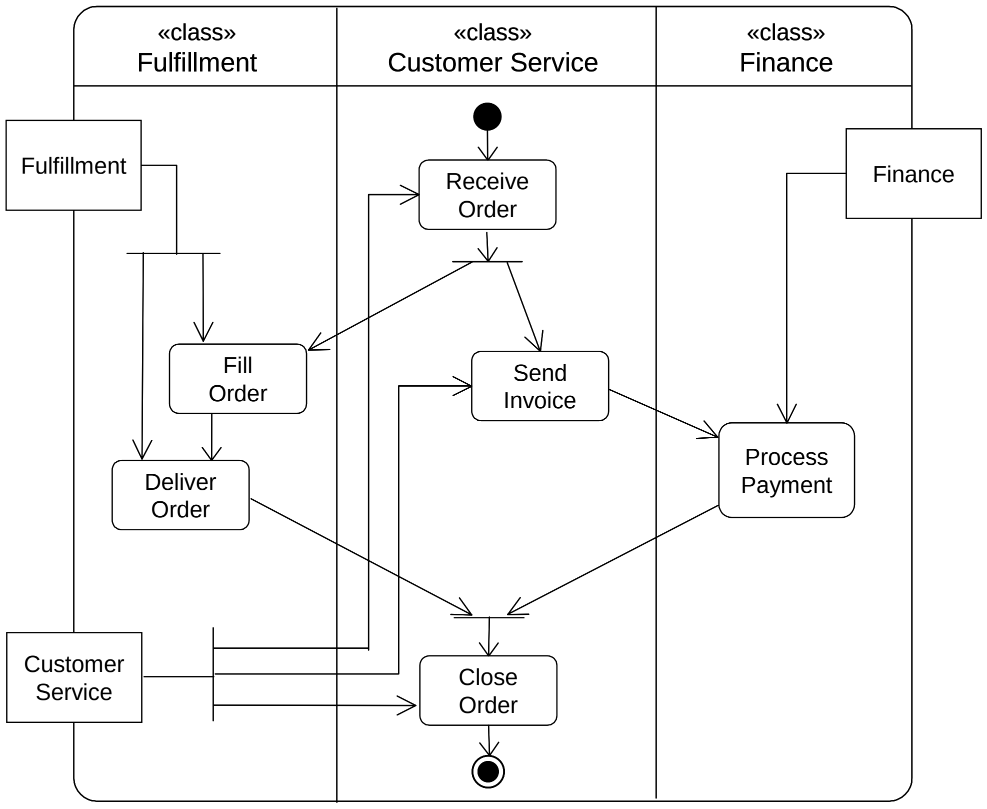


Figure 5: Partitions Representing Classes, with Object Flows

[5] An example repository model of CALLOPERATIONACTION is shown in Figure 8 of [4].

A more concise way to specify the target instances of CALLOPERATIONACTION is by navigation along attributes or associations from the same root object. Figure 6 shows an activity with partitions representing navigation from instances of class COMPANY, which has its own superpartition above the others. The navigated associations are shown in the subpartitions (in UML 2 association ends can be properties of the class from which they navigate).[6] The class model is shown in Figure 7. For each instance of COMPANY, navigating along the links FULFILLMENT, CUSTOMERSERVICE, and FINANCE will give the target instance for operation calls contained by the corresponding partition. This technique assumes that the entire activity is executed in the context of a single instance, for example as a method. Navigation proceeds from the context instance to the required targets of CALLOPERATIONACTION.[7] An alternate notation is shown in Figure 8.[8]

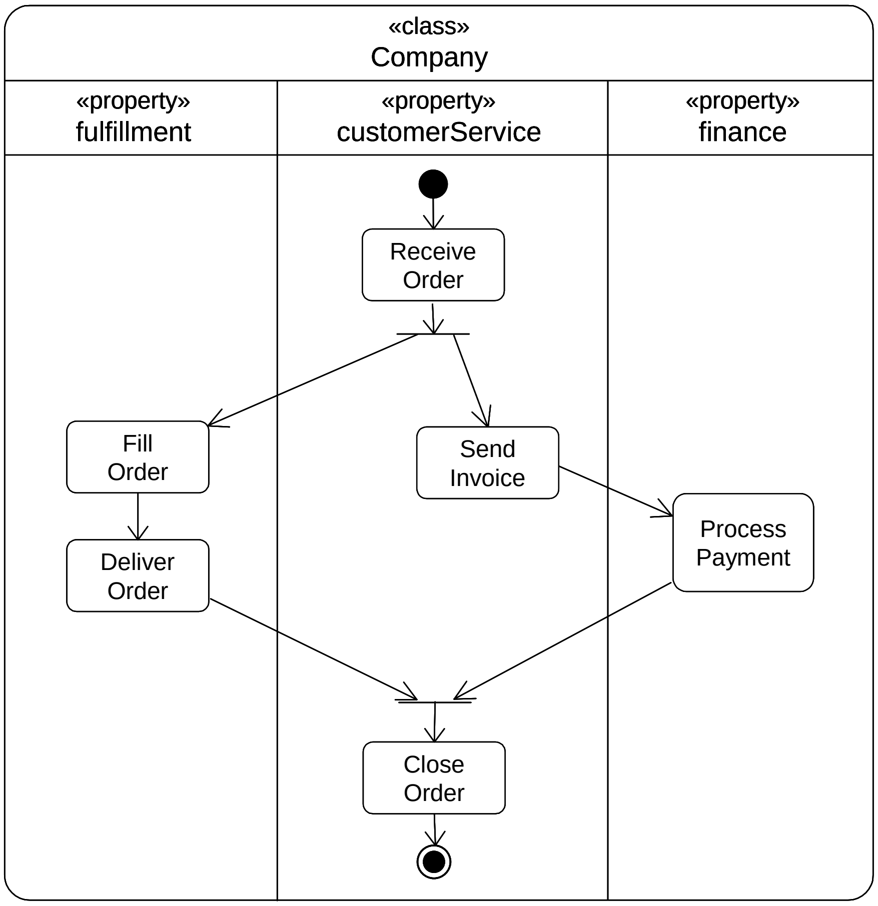


Figure 6: Partitions Representing Properties

[6] A completely class-based decomposition would show just the classes that are navigated through, for example, COMPANY and FULFILLMENT, but this will not tell exactly which instances are the targets of messages, unless the type of the properties are unique in the root class.
[7] Identifying instances by navigating from the same object is the basis of UML 2 composite structure model. This will be covered in a later article.
[8] A tool vendor could hardly be blamed for replacing double colons with a dot notation for nested navigation partitions.

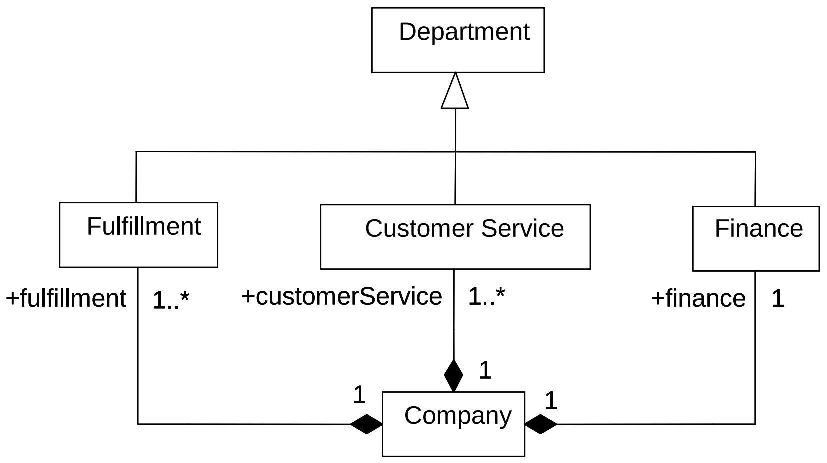


Figure 7: Class Model for Partitions in Figure 6

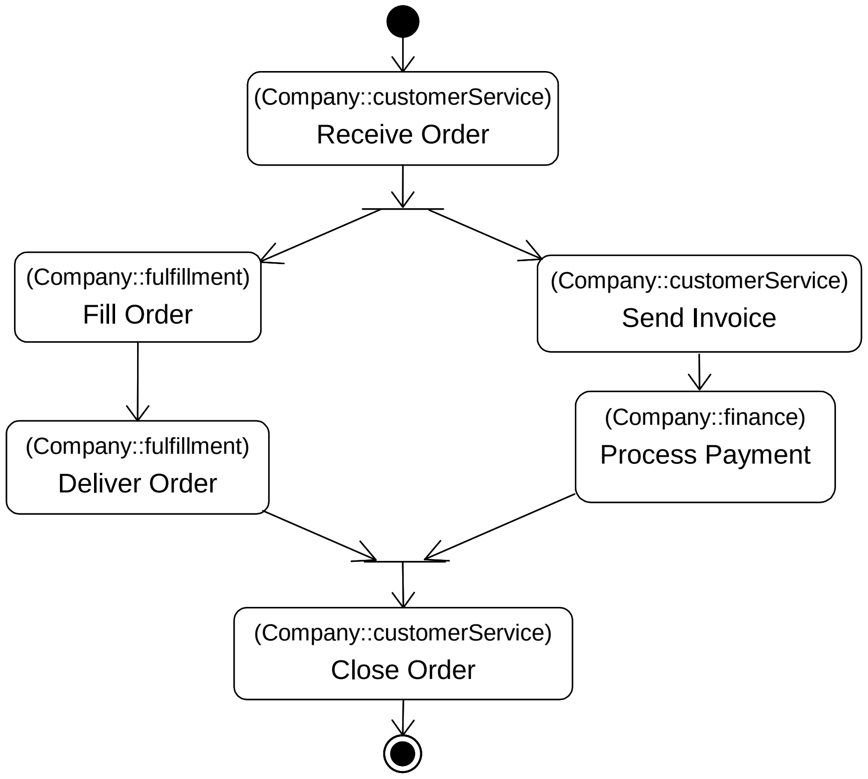


Figure 8: Partitions Representing Properties, Node-based Notation

Tools can add value to property subpartitions by automatically keeping the partitions consistent with the executable model. For example, they can generate the flows and navigation needed to specify the inputs of CALLOPERATIONACTION, as shown in Figure 9, while still presenting Figure 6 as the modeler's view. Figure 9 assumes the activity is a method on the COMPANY class that has a parameter that is bound to the instance of COMPANY on which the method is invoked. This is shown as the COMPANY activity parameter node on the upper left. The various departments are retrieved from that instance with GETSTRUCTURALFEATUREACTIONS and passed to the actions needing them.

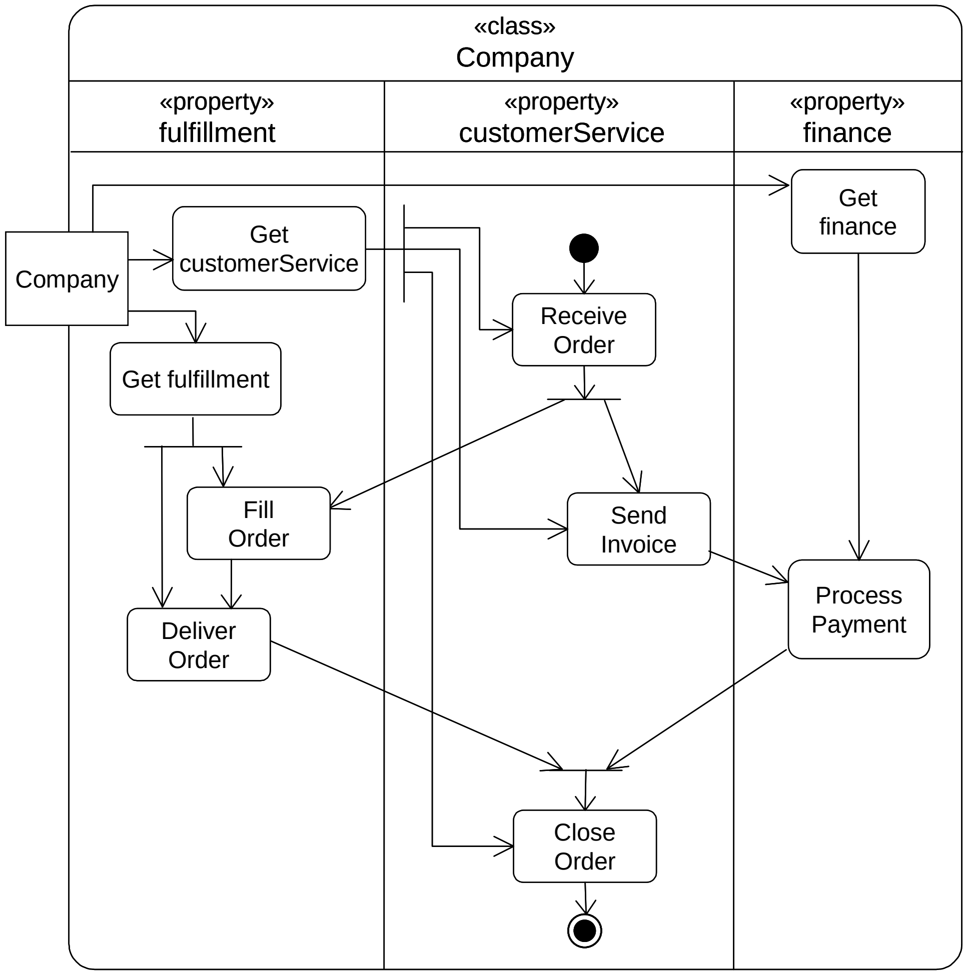


Figure 9: Partitions Representing Properties with Flows

Figures 3 and 6 refine Figure 1, but they do not dictate when refinement happens or if it happens at all. Figure 1 could be used for a long period even without the partitions before details are worked out. This facilitates application of UML by modelers who do not use object orientation (OO) routinely, such as system engineers and enterprise modelers, and provides them a path to incrementally adopt OO as needed [2]. The flexibility to combine OO with domain-specific approaches considerably widens and integrates the potential applications of UML.

3 RELATION TO INTERACTION DIAGRAMS

Activities are an abstraction of the many ways that messages pass between objects, even with class and property partitions. In particular, the edges in an activity diagram, notated by arrows, can translate to one or many messages between objects, or none at all. This makes activities useful at a stage of development where the primary concern is dependency between tasks, rather than the protocols between objects. When messages are the focus of development, UML interaction diagrams are more appropriate. These have a very different semantics from activities, even though they use similar notations, such as arrows and rectangles. Interactions also define a kind of activity notation that is overlaid on the underlying interaction model, called the *interaction overview diagram*, which has a different semantics from activities, too.

The most important difference between activities and interactions is that edges connecting actions only indicate one action starts after another completes[9], they are not messages[10]. They can be translated to messages, but this involves more than the connected actions. For example in Figure 3, the edge between SEND INVOICE and PROCESS PAYMENT means that processing payment happens after sending an invoice. It does not necessarily imply a message sent between CUSTOMER SERVICE and FINANCE, as shown in Figure 10. In particular, the SEND INVOICE behavior cannot pass a PROCESS PAYMENT message to FINANCE, because SEND INVOICE is complete before PROCESS PAYMENT starts. This ensures that SEND INVOICE is reusable in other activities without restricting what happens before and after it. For example, the customer service department may be involved in other activities that send invoices but have different actions before and after it.

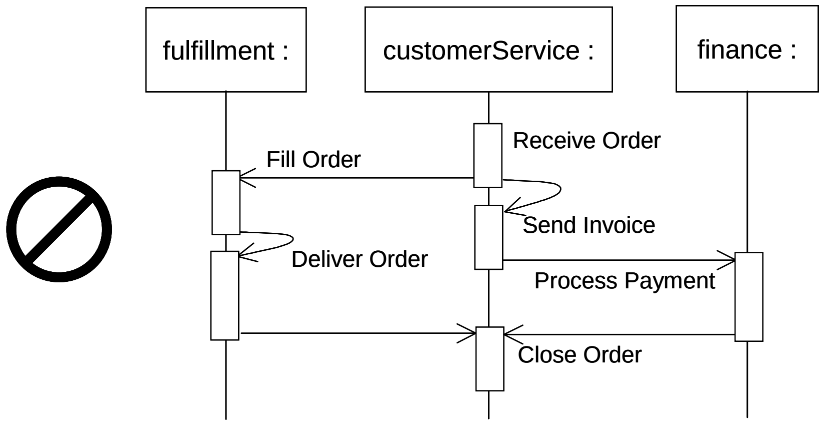


Figure 10: Interaction Diagram that is Inconsistent with Figure 6

[9] Except when used with streaming or optional parameters [4].
[10] This is a common misinterpretation of activities with class partitions, and has significant impact on methodologies that use both diagram types [7].

There are many ways that ordering of action execution might be achieved in a message passing implementation.[11] Figure 11 shows one possibility for the activity of Figure 6.[12] It uses a coordinator object to enforce the execution sequence, for example, a customer resource management system. The diagram says that the coordinator sends a RECEIVE ORDER message to the fulfillment department and after that is done, it sends other messages in parallel as shown, and when those are complete it sends the CLOSE ORDER message.[13] Alternative implementations could have one of the objects from the activity partitions coordinate everything, such as CUSTOMER SERVICE. Or all three objects could take on some portion of the process, such as FULFILLMENT coordinating FILL ORDER and DELIVER ORDER, as shown in Figure 12.[14,15,16]

[11] This is called *choreography* in web services [8].

[12] UML 2 interactions only use the navigation style shown in Figure 11, that is, they assume the targets of messages are found by navigating from a single instance. This is shown with a colon notation in the rectangle at the top of each lifeline. The name before the colon is the property name, the name after is the type of values the property holds, which is omitted in Figure 11.

[13] Interaction diagrams usually omit non-message actions, such as getting and setting attribute values, so there may be actions occurring between messages that are not shown on the diagram. Activities cannot omit steps in a sequence.

[14] In web service choreography, the processes inside interacting objects are private if the objects are independent companies. Figure 12 would be more typical of these applications. For example, the process internal to fulfillment might be hidden, and defined as its own activity diagram. Service-oriented architectures are highly decentralized in this respect. However, participants still agree on the pattern of public interactions between them, and must track where they are in these interactions in order to respond to each other properly. This is called *correlation*.

[15] The curved arrows in the interaction figures are a UML 1.x notation indicating an object that is sending a message to itself. It is not clear from the specification whether this carries over to UML 2, and will be addressed in finalization.

[16] The UML 2 communication diagram, which was called the *collaboration diagram* in UML 1.x, augments interaction diagrams with the connections between objects that are used to determine the targets of messages. Messages are shown passing along these connections, with numbers to show ordering. The relation between these diagrams is being clarified by the response to UML for Systems Engineering [9].

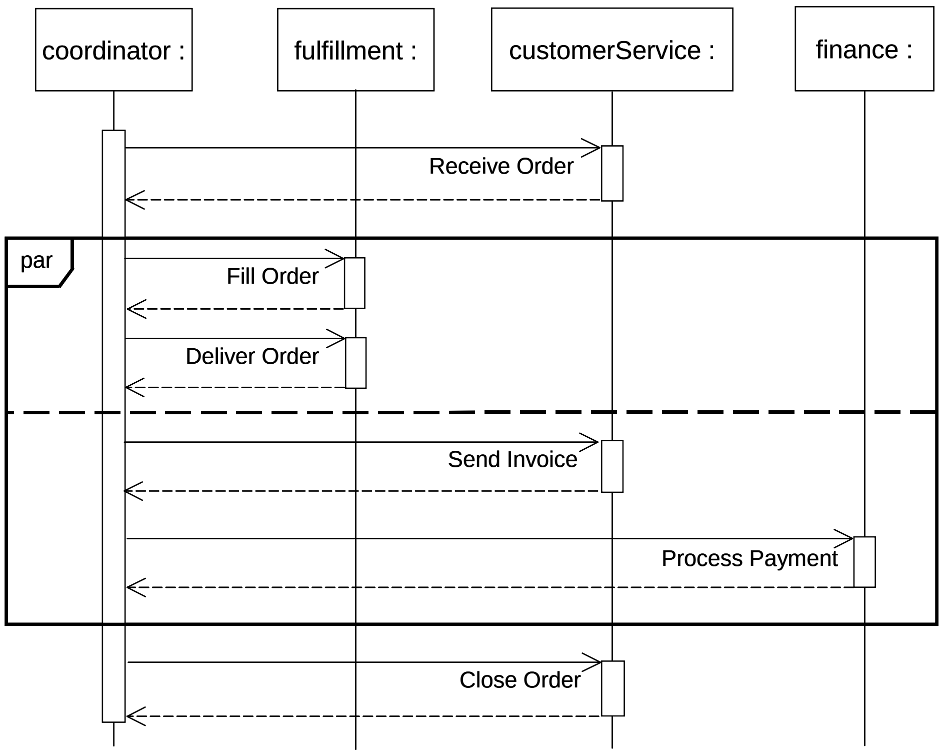


Figure 11: Possible Interaction Diagram for Figure 6

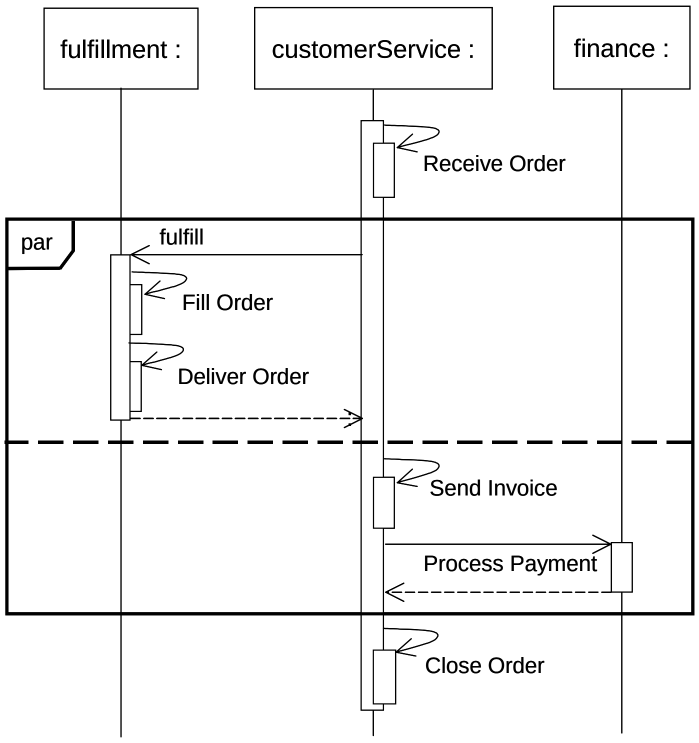


Figure 12: Another Interaction Diagram for Figure 6

The interaction overview diagram is an alternative notation for interactions that looks similar to activities, as shown in Figure 13, but is stored as an interaction in the repository. Each "action" can show a message or messages, or refer to an entire interaction. It is a way to highlight the control aspects of an interaction, but has the disadvantage of being large and hard to draw in some cases. The semantics is completely defined by interactions, rather than activities. For example, the rectangles labeled *ref* mean that the interaction named in the rectangle is copied in at that point, like a programming macro, rather than invoking a behavior as an action would.

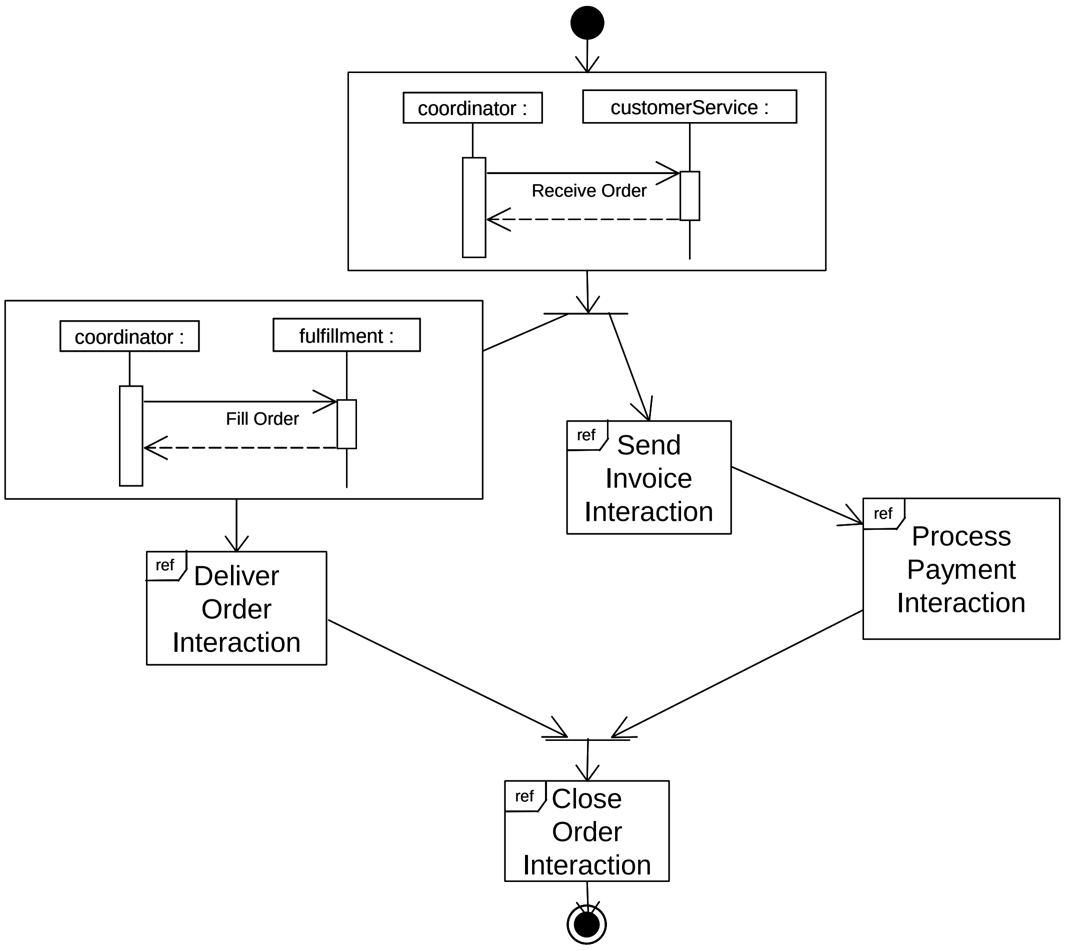


Figure 13: Interaction Overview Diagram

4 BEHAVIOR PROPERTY PARTITIONS

Behaviors in UML 2 are also classes, and their instances are running executions of the behaviors [2]. Behavior instances can carry information about executions, such as how long they have been running, what resources they have locked, as well as operations, such as suspend and resume. Partitions can specify values for executing behaviors. For example, Figure 14 shows the location where each behavior is performed (turned sideways for example in the next section). The property PERFORMINGLOCATION is a modeler-defined property, and can have values defined for its type, LOCATION, as shown by the class model in Figure 15. The COMPANYBEHAVIOR class defines a property to inherit to the various methods implementing the operations in Figure 4. The partitions of Figure 14 indicate the values of PERFORMINGLOCATION of the executing methods. The values are not assigned in the class model, because the methods may be used in other activities requiring different locations for the behavior executions.

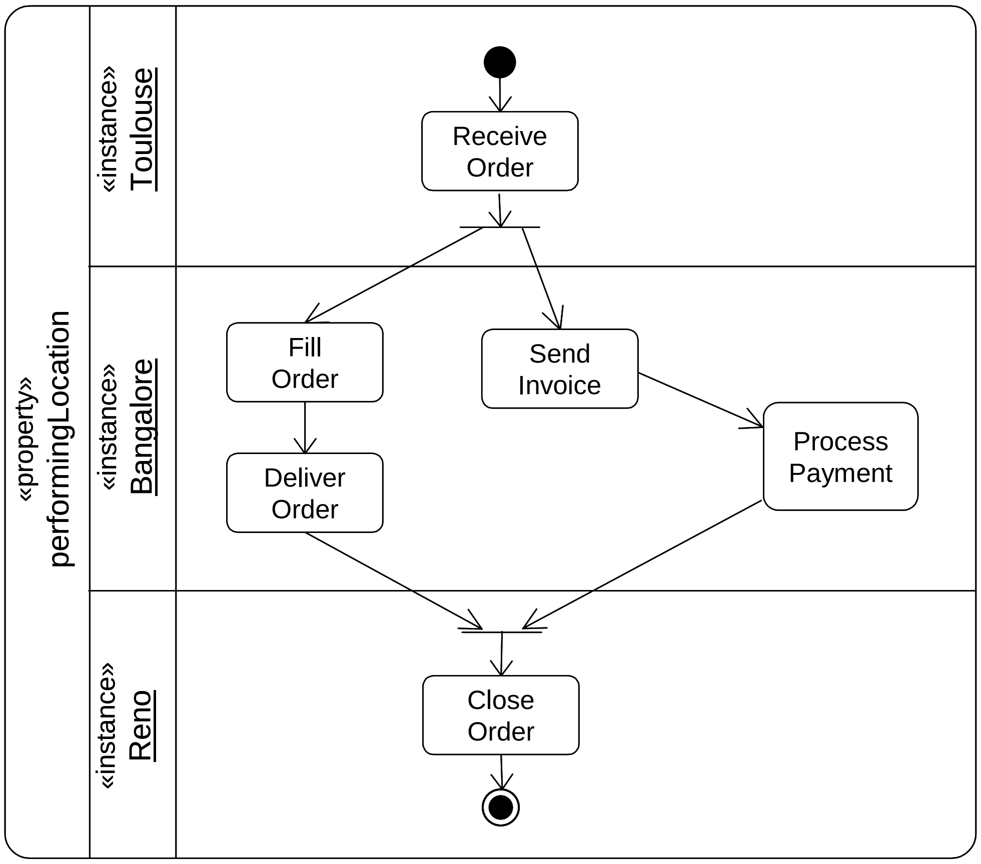
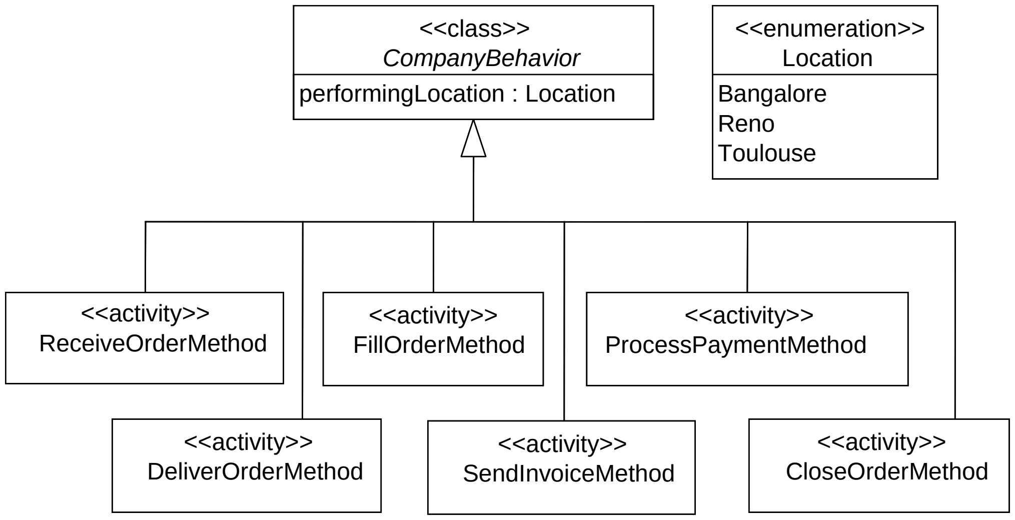


Figure 14: Partition as Attribute Value



<<class>>
CompanyBehavior
performingLocation : Location
<<enumeration>>
Location
Bangalore
Reno
Toulouse
<<activity>> ReceiveOrderMethod
<<activity>> FillOrderMethod
<<activity>> ProcessPaymentMethod
<<activity>> DeliverOrderMethod
<<activity>> SendInvoiceMethod
<<activity>> CloseOrderMethod

Figure 15: Class Model for Figure 14

5 STRUCTURED PARTITIONS

One way to structure partitions is to nest them, as already shown in Figure 6.[17] Another mechanism is to use more than one dimension, as shown in Figure 16, which combines Figures 6 and 14. Tools can add value to multidimensional partitions by allowing some to be hidden, and supporting different node positions for each dimension, if the application does not require showing them all at once. The node-based notation is shown in Figure 17.

[17] Nested partitions can also represent nested classes.

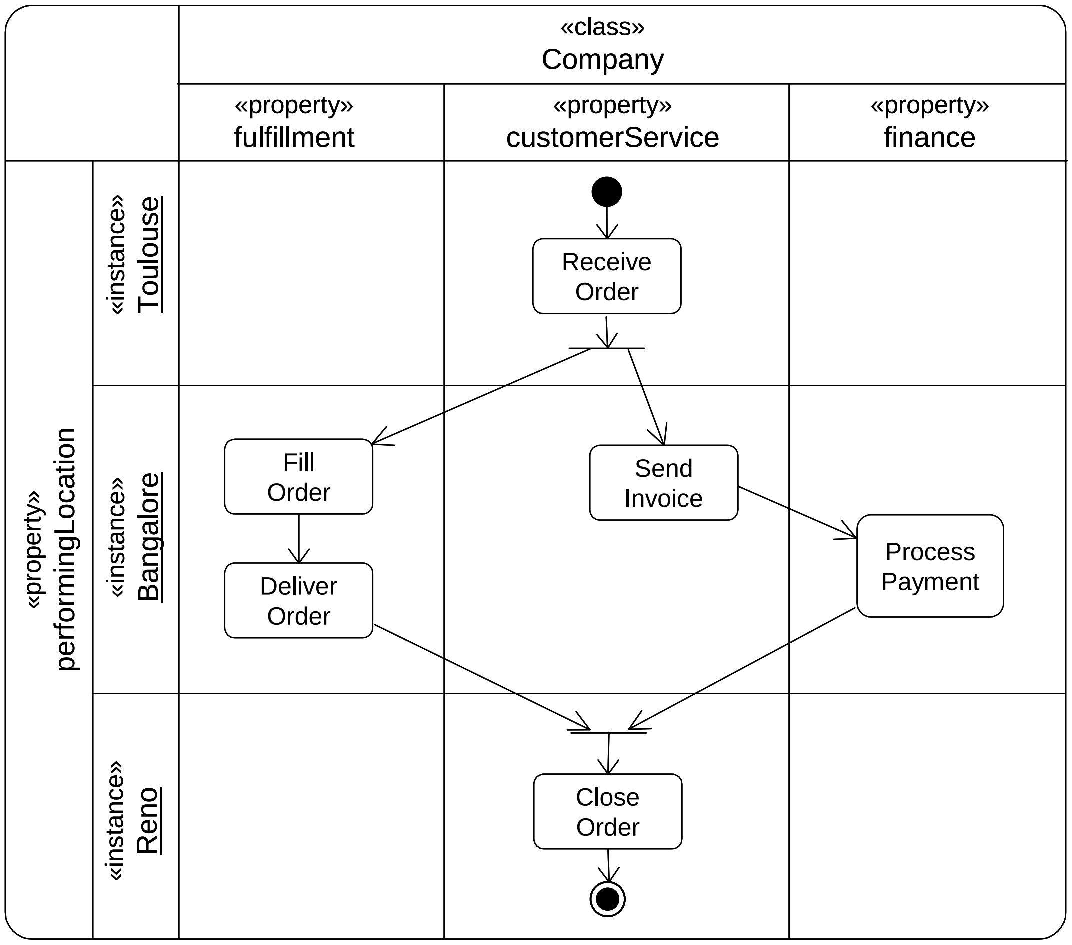


Figure 16: Multi-dimensional Partitions

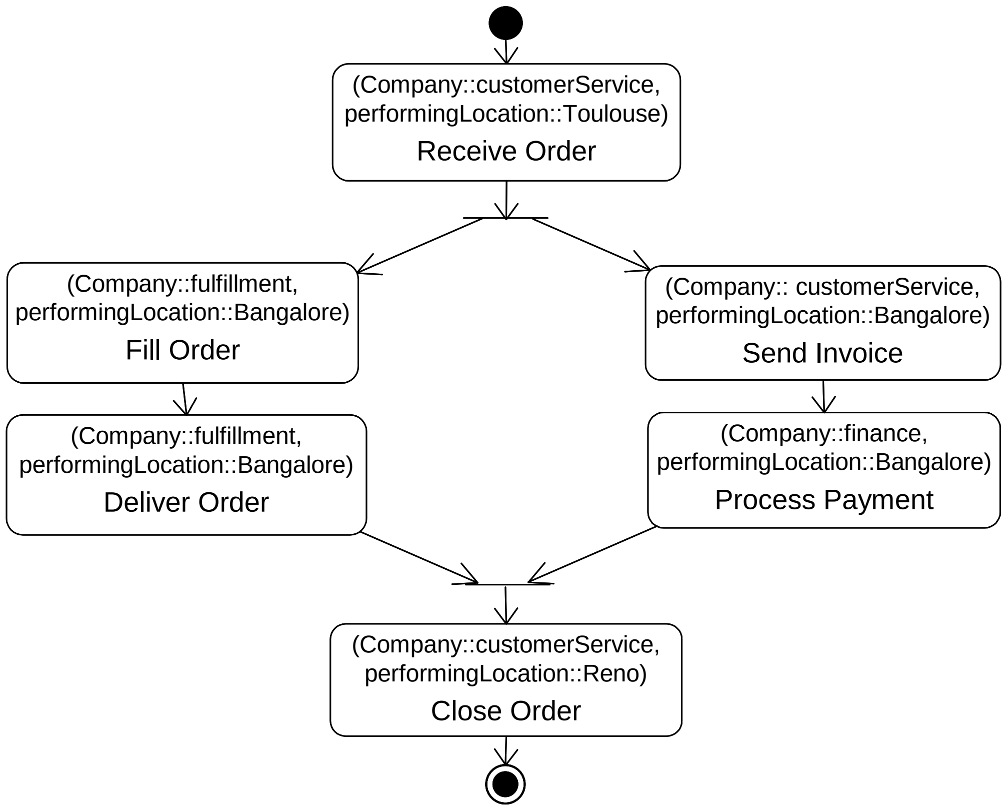

(Company::customerService, performingLocation::Toulouse)
Receive Order
(Company::fulfillment, performingLocation::Bangalore)
Fill Order
(Company:: customerService, performingLocation::Bangalore)
Send Invoice
(Company::fulfillment, performingLocation::Bangalore)
Deliver Order
(Company::finance, performingLocation::Bangalore)
Process Payment
(Company::customerService, performingLocation::Reno)
Close Order

Figure 17: Multi-dimensional Partitions, Node-based Notation

A partial repository model for Figures 3, 15, and 16 is shown in Figure 18.[18] The top-level partitions are marked as dimensions, otherwise the repository cannot tell they are drawn at different angles. The model shows a CALLOPERATIONACTION for FILL ORDER and the partitions containing the action. The partitions at the bottom of the figure for COMPANY and FULFILLMENT indicate that the operation FILL ORDER must be owned by the type of the FULFILLMENT property, namely the FULFILLMENT class. The partitions at the top of the figure for PERFORMINGLOCATION and BANGALORE indicate that the execution of the method dispatched by FILL ORDER must be performed in Bangalore. The location property is inherited from COMPANYBEHAVIOR.

[18] The GENERAL association is derived from an additional metaclass for generalization that is not shown. The SUBGROUP association from partition to the partitions it contains should be named SUBPARTITION for consistency. This will be addressed in finalization.

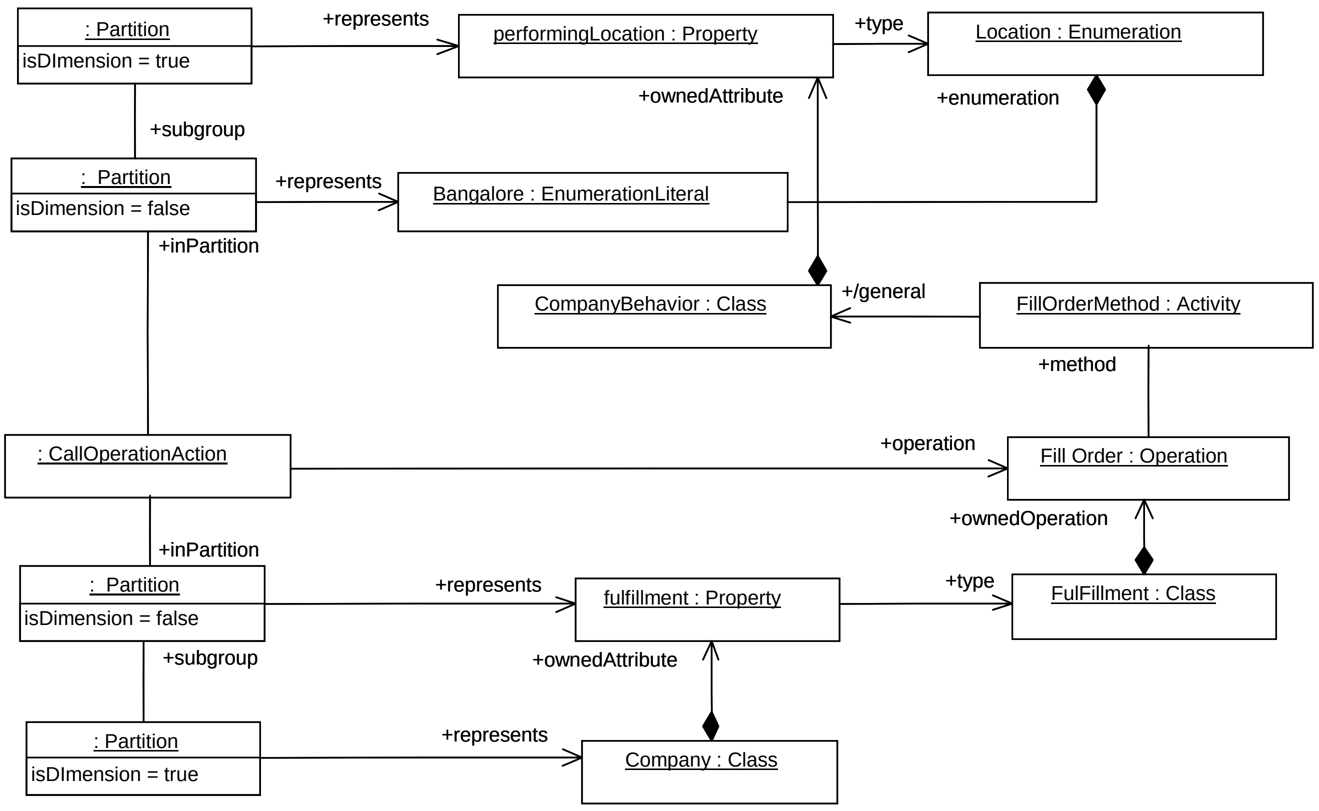


Figure 18: Partial Repository Model for Figures 3, 15, and 16

Partitions can be marked as external to the pattern set by the others in the same dimension, as shown in Figure 19. In this example, the external partition for CUSTOMER is not a part of the company, though presumably there is some navigation from the company to the customer. When using external partitions with multiple dimensions, the repository stores a hidden dimension partition over the external partition and its sibling, which is not shown in the diagram. For example, the COMPANY and CUSTOMER partition will be in a larger partition that is marked as a dimension. Otherwise, the repository cannot tell which dimension the external partition belongs to.

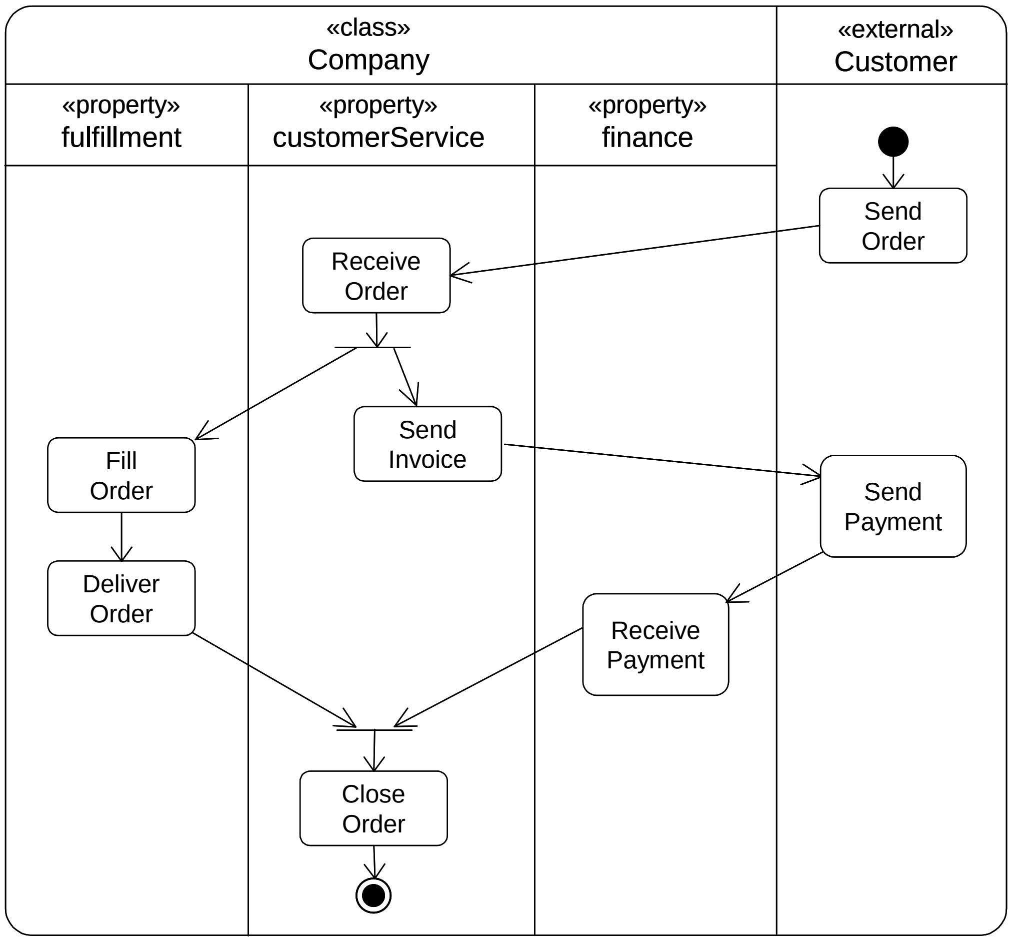


Figure 19: External Partition

6 MODELER-DEFINED PARTITION PATTERNS

The uses of partitions so far are defined in UML, but modelers can also define their own. Figure 20 shows an example of partitions representing states. An instance of ORDER moves through various states based on progress through the activity. There are many state machines corresponding to the activity, partly due to the many possible messaging implementations, as explained in section 3. The state machine for ORDER contains the states given in Figure 20, but the transition triggers between them would depend on whether the order or some other object is coordinating the actions, or both. For those actions coordinated by the order, there are more variations depending on whether this is done by an activity or a state machine. Assuming states coordinate the actions, as in the UML 1.x version of activities, the translation from the activity Figure 20 must account for differences in concurrency semantics between activities and state machines [5]. The one-to-many relation of activities to state machines is another example of the abstraction that activities provide for object implementations.

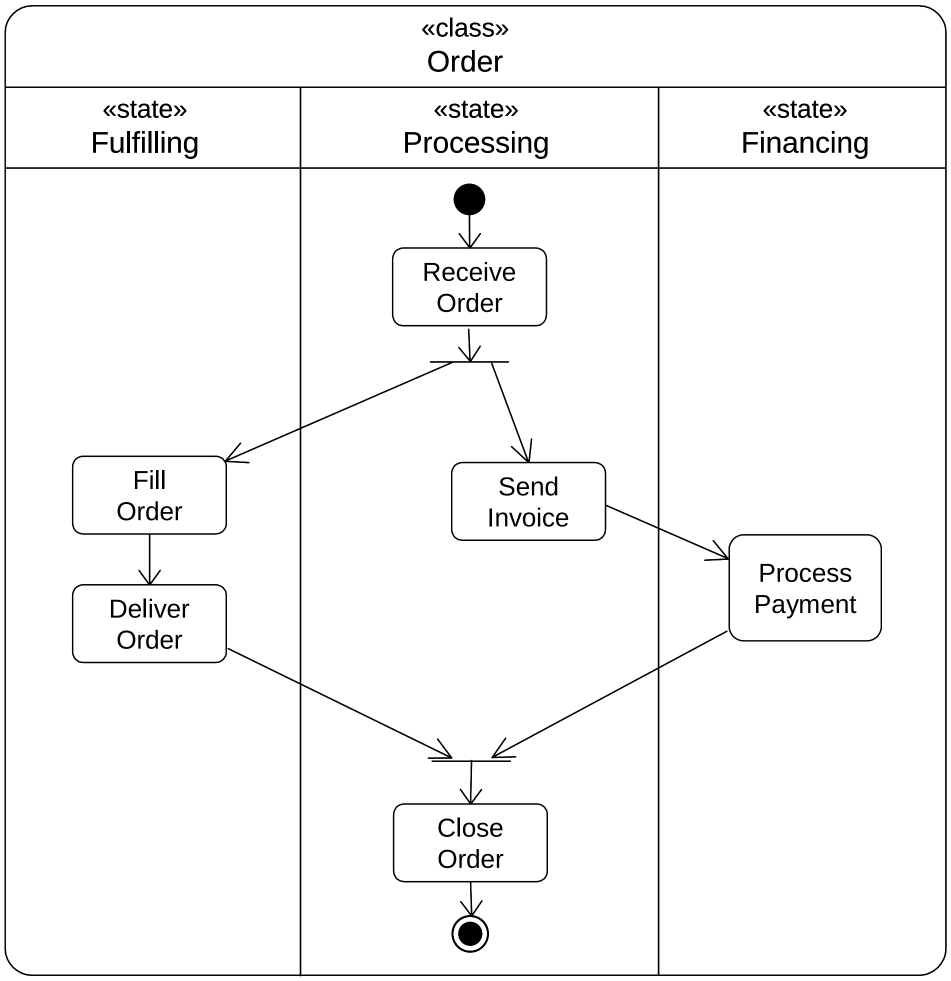


Figure 20: Modeler-defined Partition Pattern

7 CONCLUSION

This is the fifth in a series on the UML 2 activity and action models. It covers partitions, which are a way of grouping actions that have some characteristic in common. Typical usage patterns are described based on the element a partition represents: classes, properties of classes, and properties of behaviors. These patterns are combined using partition nesting and multiple dimensions. Various degrees of refinement are presented, to illustrate partitions used early in development for sketching, and the transition to later stages concerned with the details of execution. Partitions usually identify the entities responsible for actions, which is an abstraction of message passing as supported by interaction diagrams. A modeler-defined usage of partitions is presented to show an activity abstraction of state machines. These examples show how activities focus on task dependency, rather than object or state dependency.

ACKNOWLEDGEMENTS

Thanks to Evan Wallace and James Odell for their input to this article.

REFERENCES

[1] Object Management Group, "UML 2.0 Superstructure Specification," http://www.omg.org/cgi-bin/doc?ptc/03-08-02, August 2003.

[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.

[3] Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, September 2003.

[4] Bock, C., "UML 2 Activity and Action Models, Part 2: Actions," in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 41-56, http://www.jot.fm/issues/issue_2003_09/column4.

[5] Bock, C., "UML 2 Activity and Action Models, Part 3: Control Nodes," in *Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 7-23, http://www.jot.fm/issues/issue_2003_11/column1.

[6] Odell, James, personal communication, 2004.

[7] Wagenhals, L., Haider, S., Levis A., "Synthesizing executable models of object oriented architectures," in *Journal of the International Council on Systems Engineering*, vol. 6, no. 4, pp. 266-300, October 2003.

[8] W3C Web Services Choreography Working Group, "WS Choreography Model Overview," http://www.w3.org/2002/ws/chor/edcopies/model/ModelOverview.html, December 2003.

[9] OMG Systems Engineering DSIG, "UML for Systems Engineering RFP," http://www.omg.org/cgi-bin/doc?ad/03-03-41, March 2003.

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 at nist.gov.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol.4, No. 4, May-June 2005

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,[1] 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

[1] 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.[2]

The dependencies of structured and flow packages are shown in Figure 1.[3] 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.

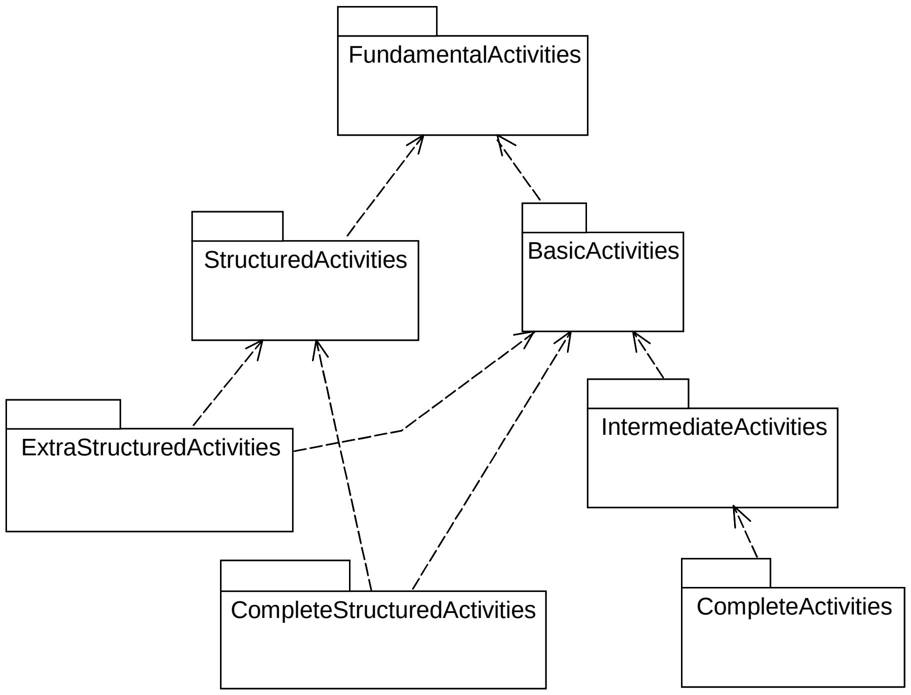


Figure 1: Activities Package Dependency

[2] 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.
[3] Compare to packaging before finalization in Figure 2 of [2].

The structured models in STRUCTUREDACTIVITIES and COMPLETESTRUCTURED ACTIVITIES 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.

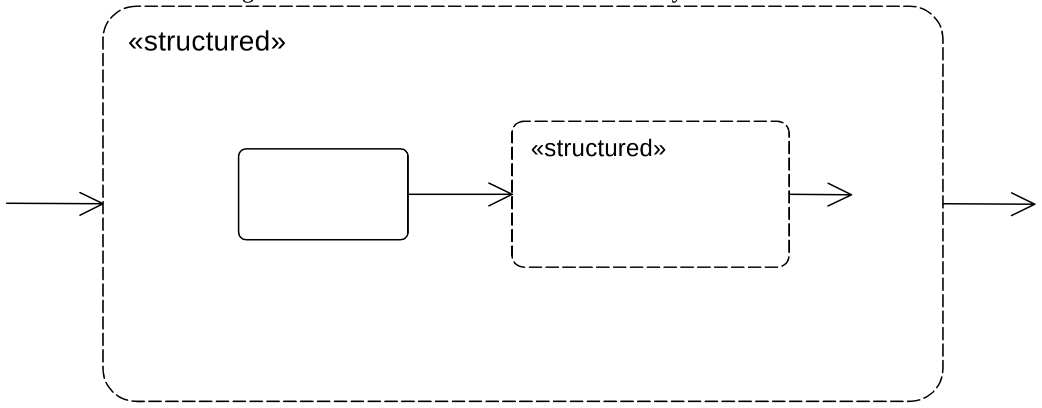
Most of the structured models do not have a standard UML notation, because they are intended for generation from various programming and action languages.[4] Examples in the following sections use the textual syntax of typical programming languages, but only as a visual presentation of the models, borrowing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

[4] 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.

Figure 2: UML Notation for Structured Activity Nodes



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.[5] 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

[5] 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

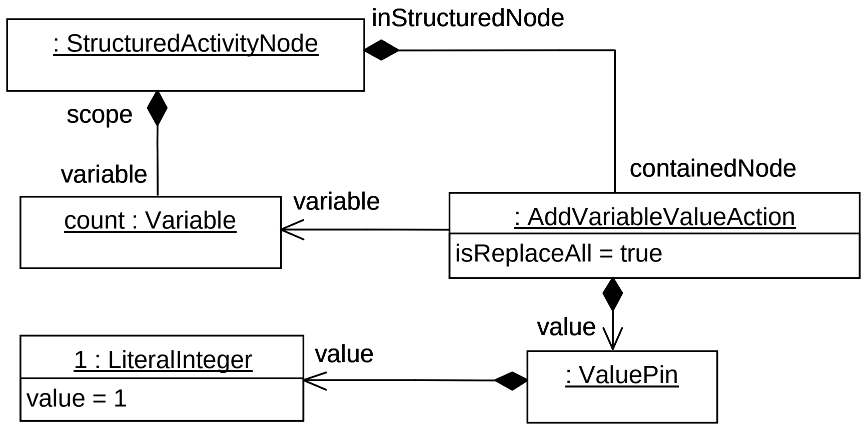


Figure 4: Repository Model for Figure 3

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.[6] 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 }`.

[6] 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.[7]

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

Figure 5: Example Textual Notation for Sequences

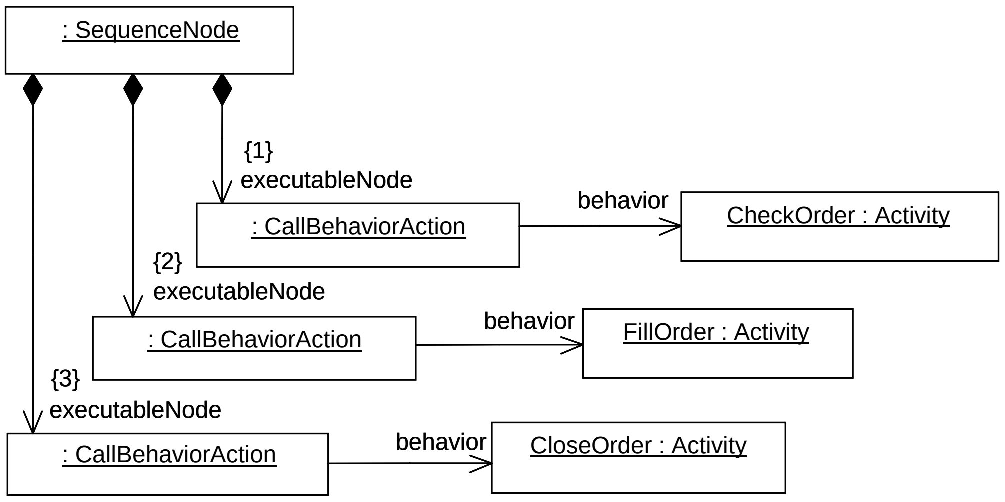


Figure 6: Repository Model for Figure 5

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

[7] 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

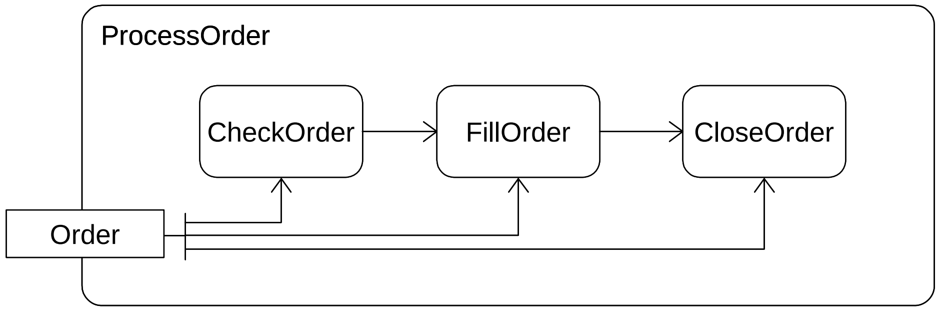


Figure 8: UML Notation for Parameters

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.[8] 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.

[8] 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

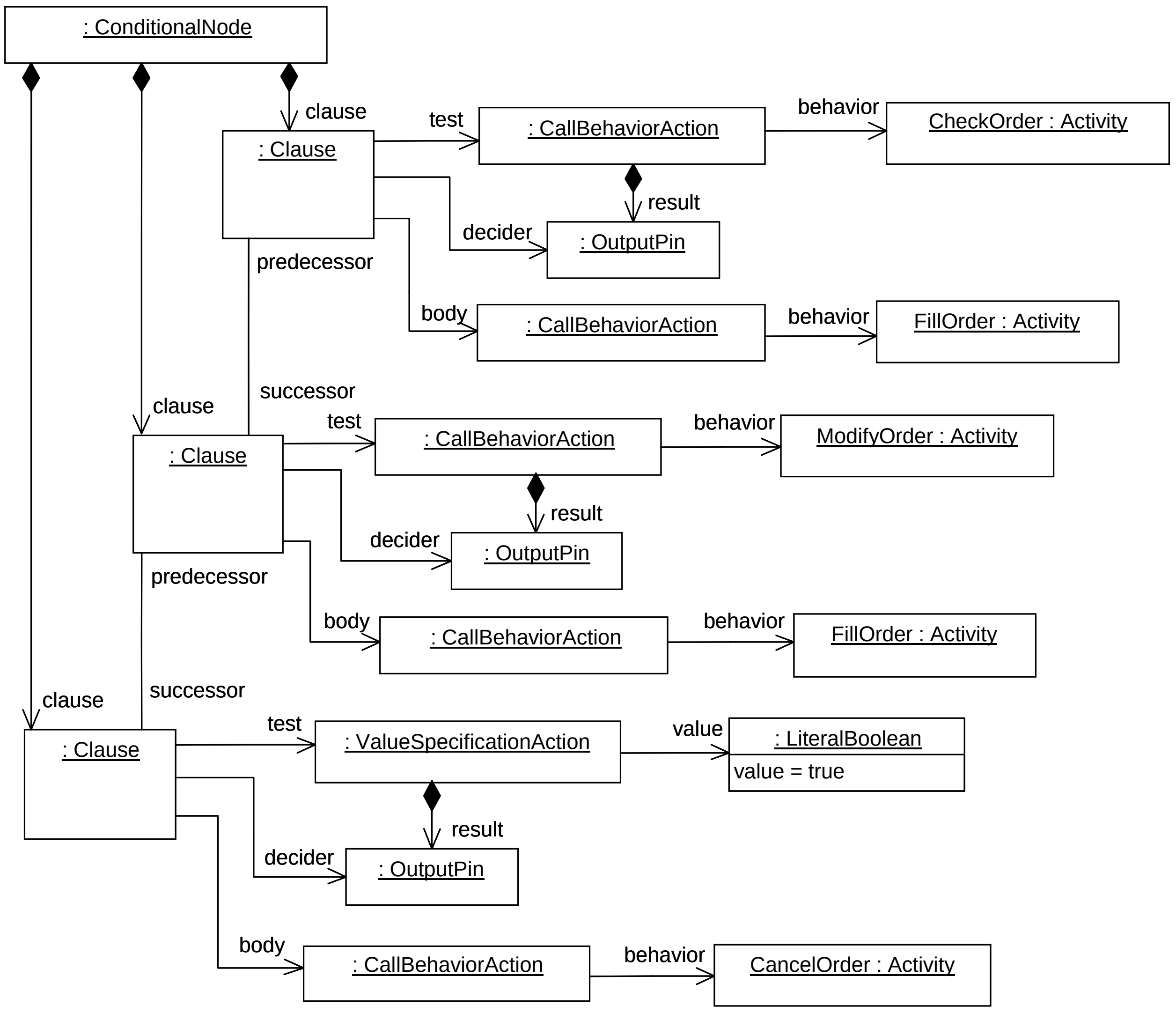


Figure 10: Repository Model for Figure 9

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.[9] 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

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

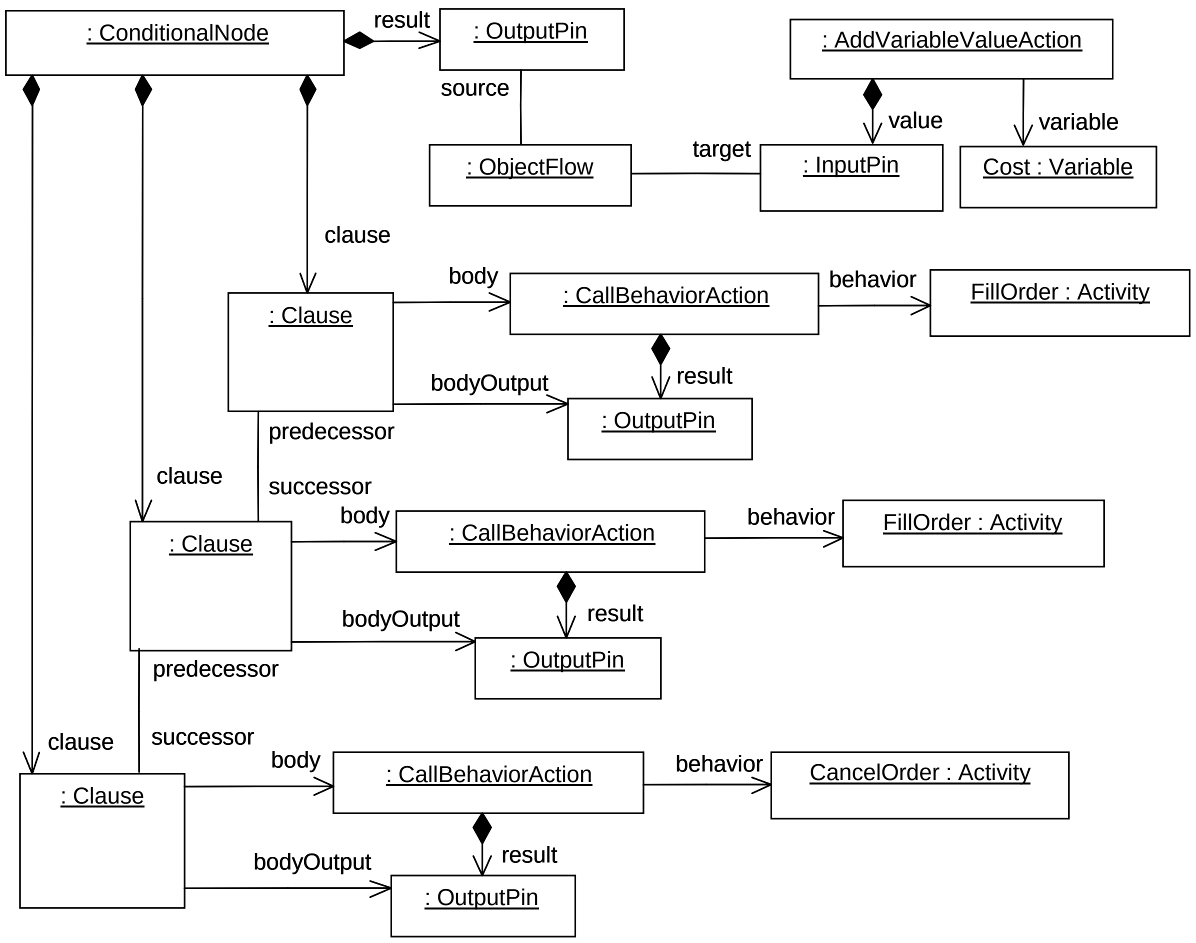


Figure 12: Repository Model for Figure 11

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.[10] 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

[10] 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.[11]

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


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

Figure 13: Example Textual Notations for Loops

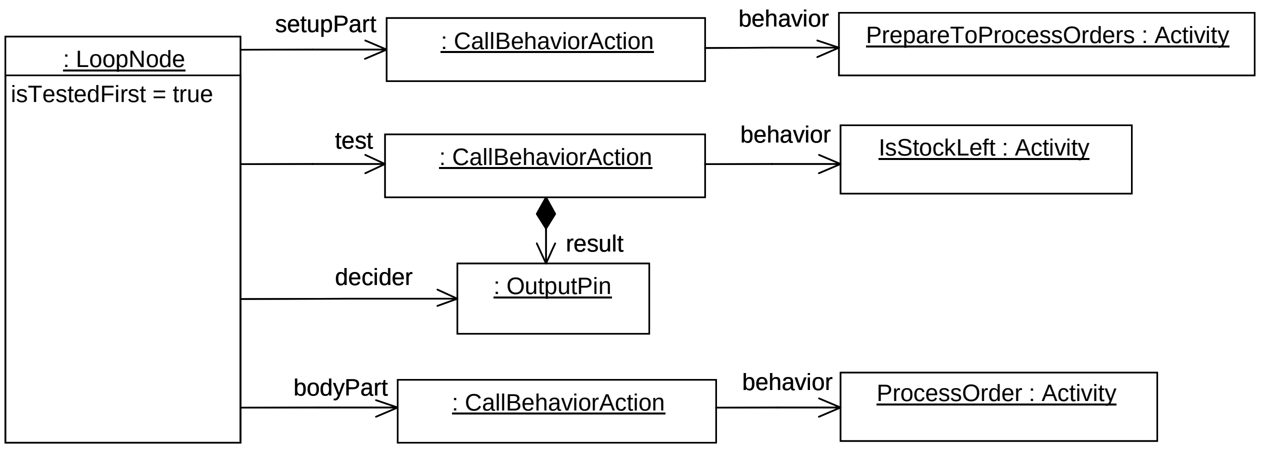


Figure 14: Repository Model for Figure 13

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 Figure4, 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.[12] The LOOPVARIABLE association specifies pins used for loop variables. In this example it is the one corresponding to AMOUNTSHIPPEDSOFAR in Figure 15.[13] 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

[11] 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.
[12] 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.
[13] 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.[14] 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

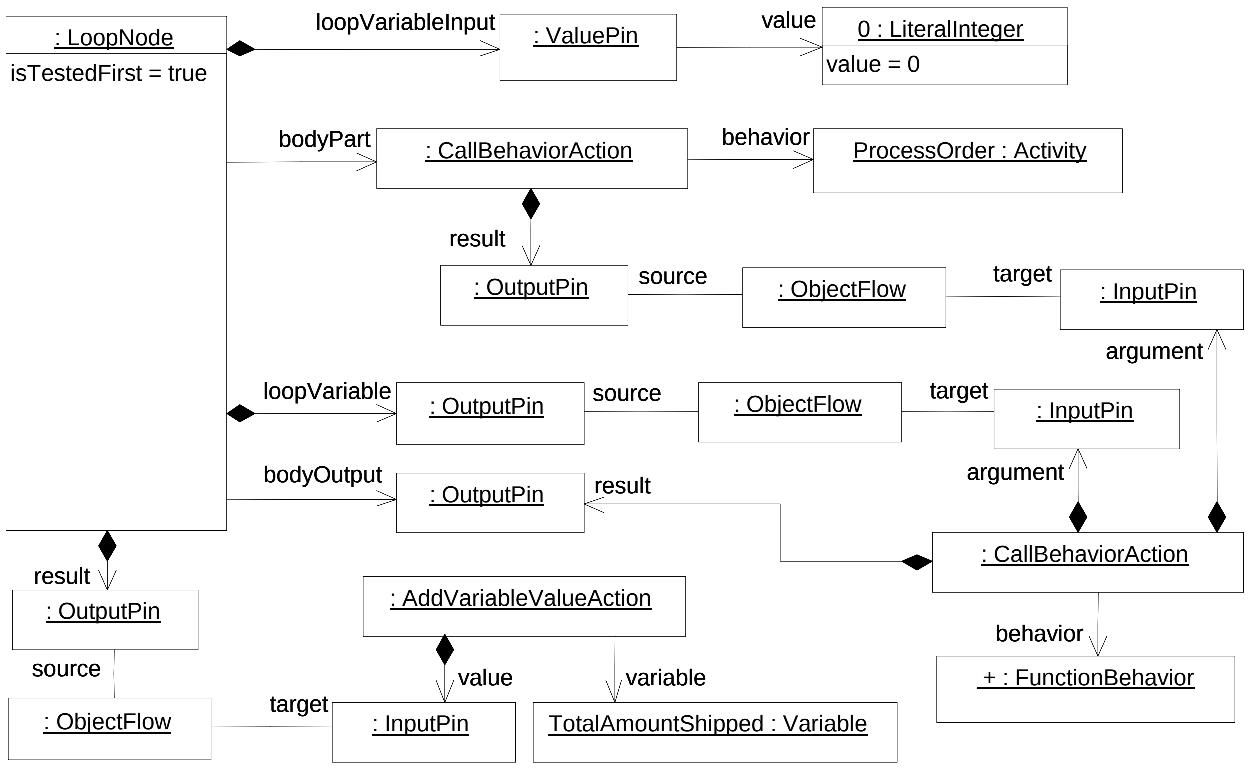


Figure 16: Repository Model for Figure 15

[14] 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.[15]

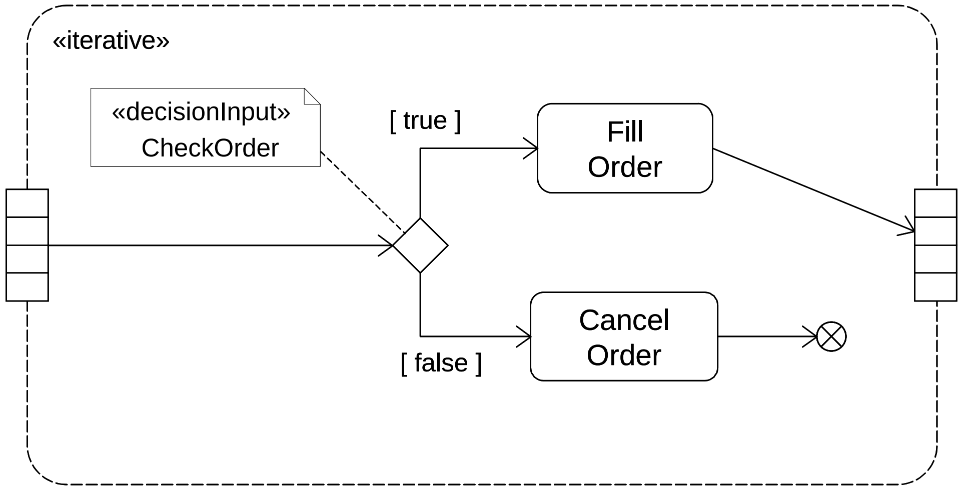


Figure 17: UML Notation for Expansion Regions

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

Figure 18: Example Textual Notation forFigure 17

[15] 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."[16] 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.[17] They can be the source and target of object flows, as all object nodes can.

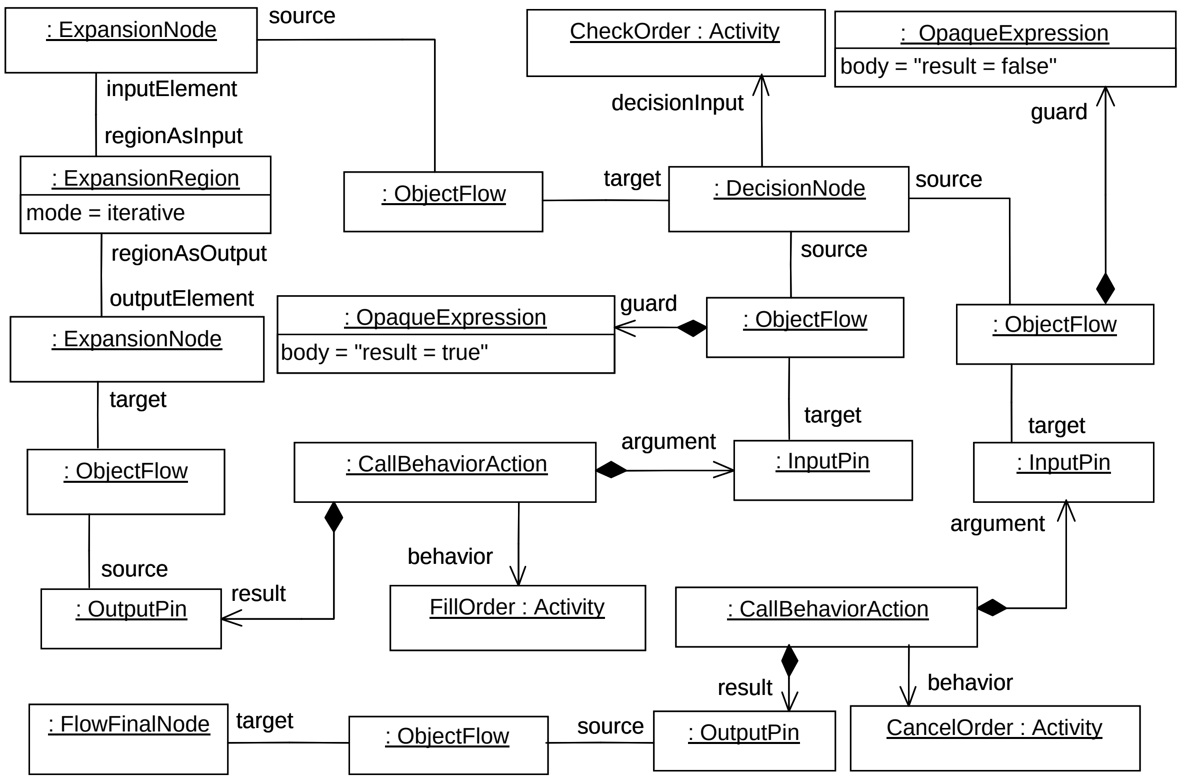


Figure 19: Repository Model for Figure 17 and Figure 18

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

[16] 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.

[17] 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.[18]

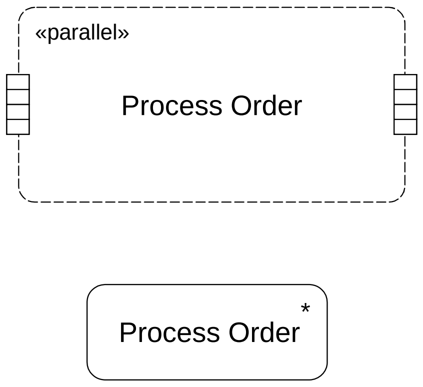


Figure 20: Shorthand UML Notations for Expansion Regions

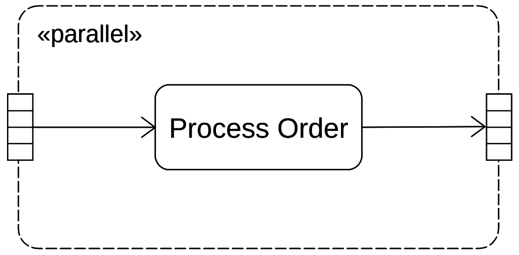


Figure 21: Longhand UML Notation for Figure 20

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."[19] Since the region

[18] 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.

[19] 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.

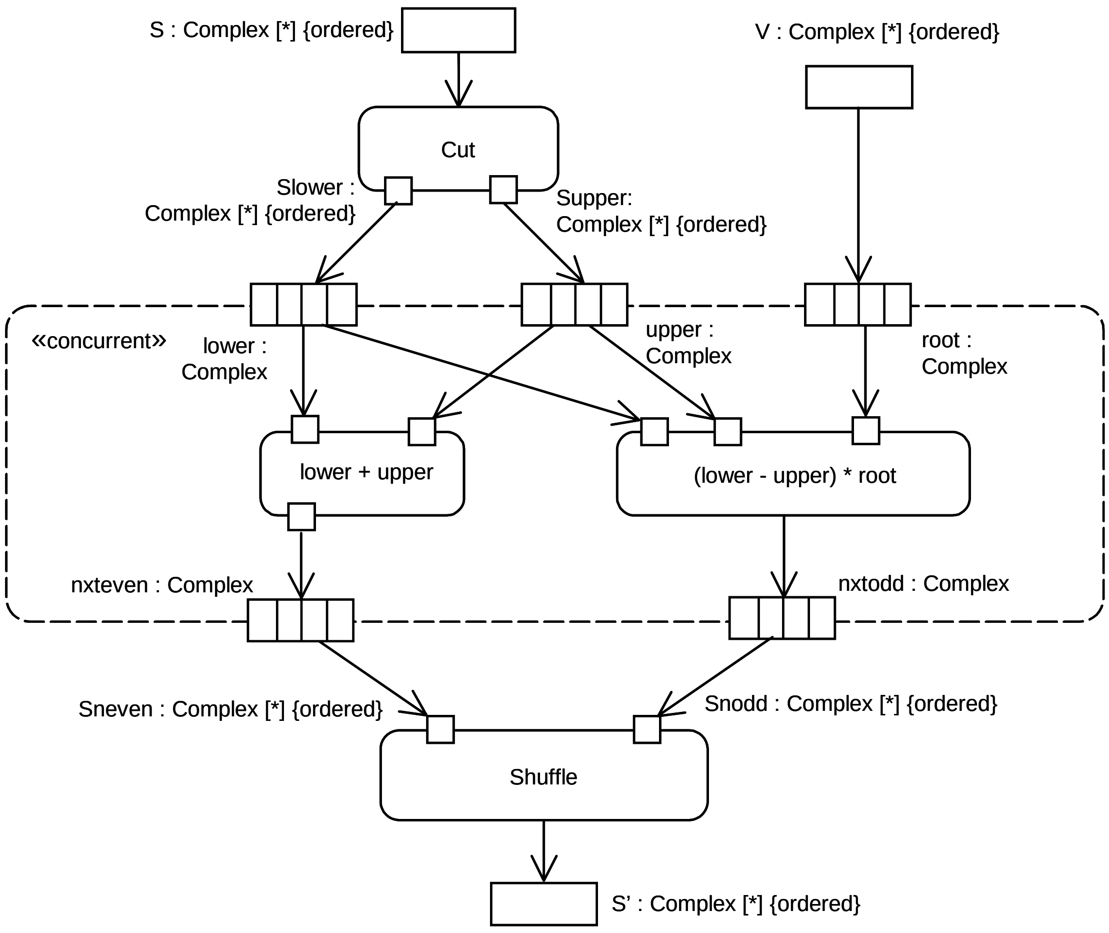

S : Complex [*] {ordered}
V : Complex [*] {ordered}
Cut
Slower : Complex [*] {ordered}
Supper: Complex [*] {ordered}
«concurrent»
lower : Complex
upper : Complex
root : Complex
lower + upper
(lower - upper) * root
nxteven : Complex
nxtodd : Complex
Sneven : Complex [*] {ordered}
Snodd : Complex [*] {ordered}
Shuffle
S' : Complex [*] {ordered}

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.[20]

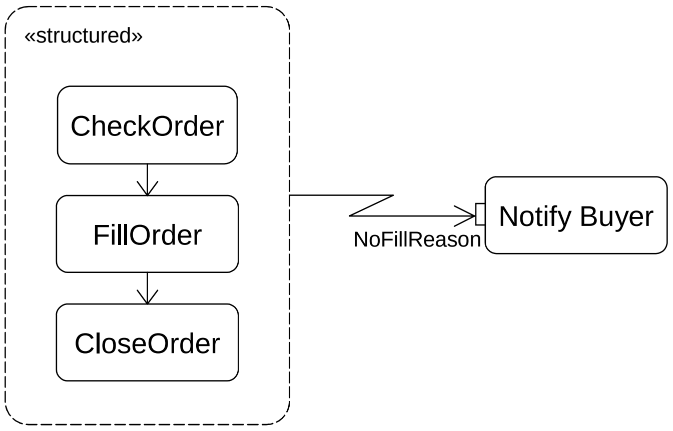


Figure 23: UML Notation for Exception Handlers

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

Figure 24: Example Textual Notation for Exception Handlers

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

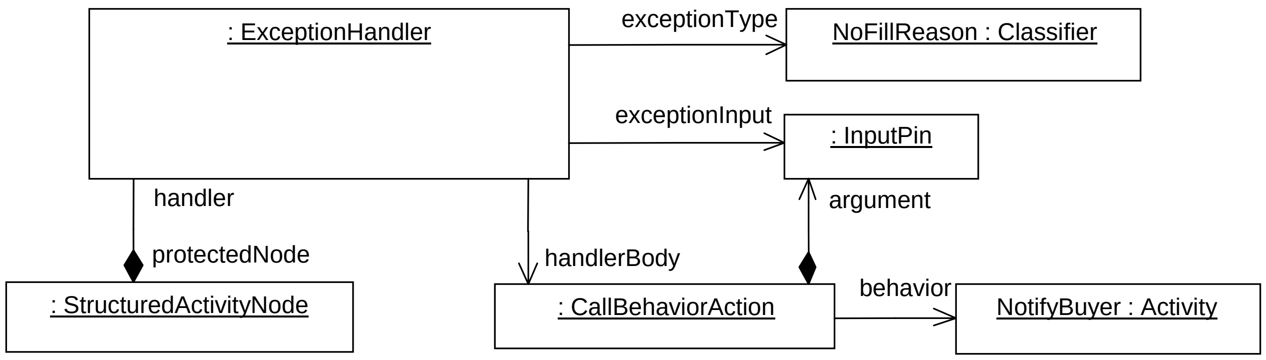


Figure 25: Repository Model for Figure 23 and Figure 24

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.[21] 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.[22]

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.[23] When a value flows along the interrupting edge, all tokens in the region are destroyed.[24] 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.[25] 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

[21] 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.
[22] 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.
[23] An alternative notation for an interrupting edge is to place a zigzag icon above a straight line. See Figure 273 of [1].
[24] The term "interruptible" region is a misnomer, because the region cannot be restarted with the same token state as when it was terminated.
[25] 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.

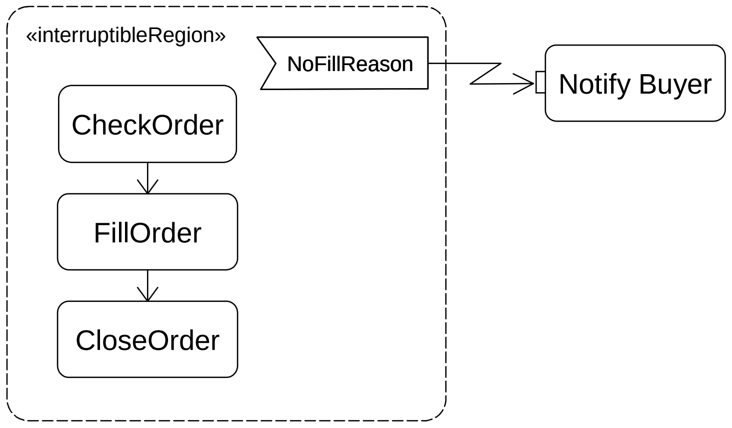


Figure 26: UML Notation for Interruptible Regions

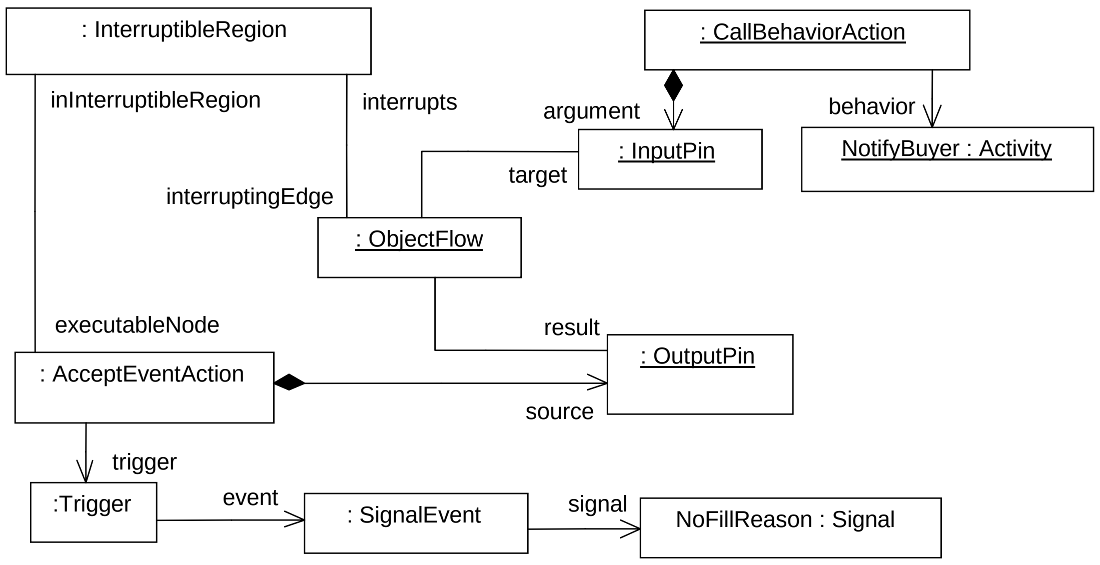


Figure 27: Repository Model for Figure 26

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].

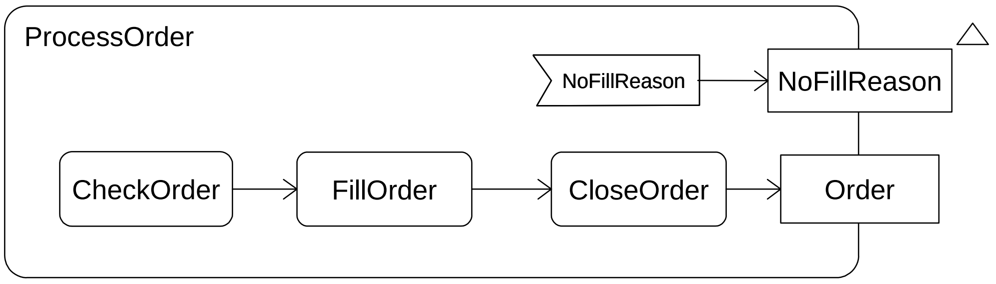


Figure 28: Exception Output Parameter

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.[26] 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

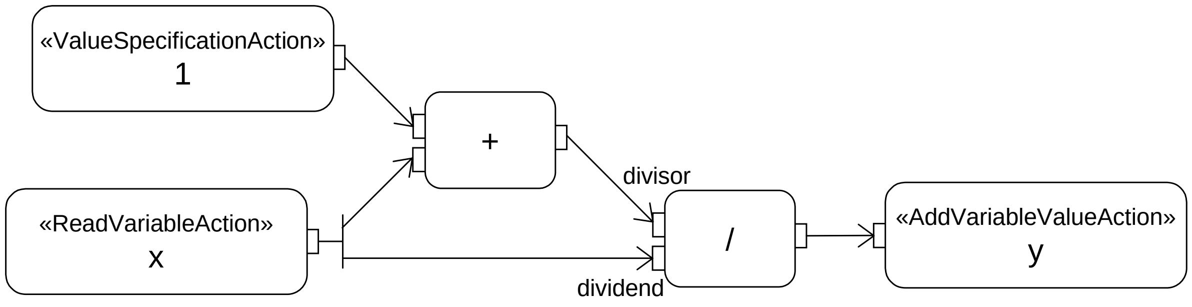


Figure 30: Equivalent UML Notation for Figure 29

[26] 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.

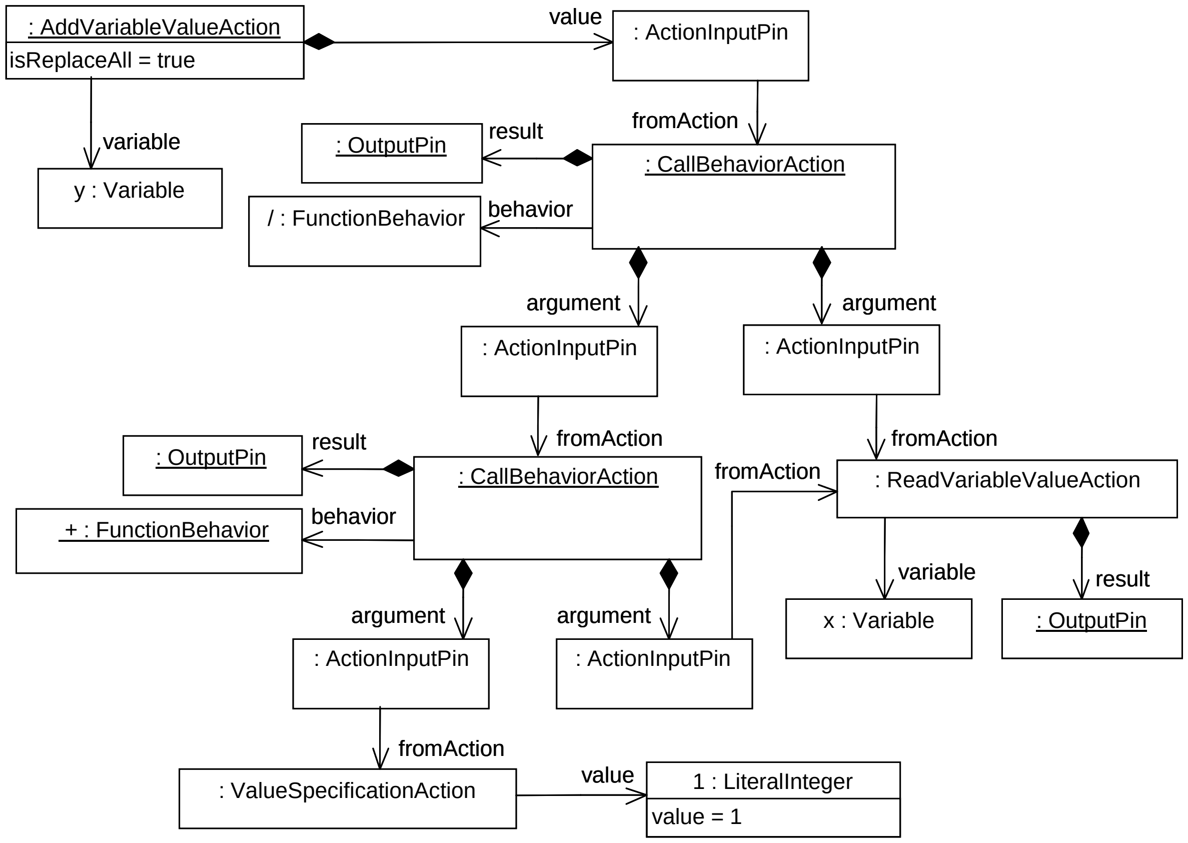


Figure 31: Repository Model for Figure 29 Using Action Pins

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

[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

[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

[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

[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 at nist.gov.