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.

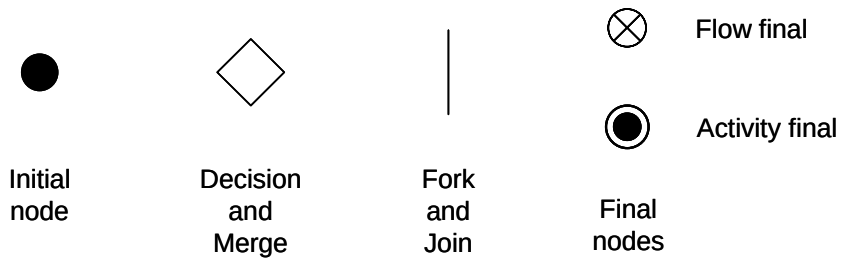


Figure 1: Control Nodes

Cite this column as follows: Conrad Bock: “UML 2 Activity and Action Models Part 3: Control Nodes”, in *Journal of Object Technology*, vol. 2, no. 6, pp. 7-23.
http://www.jot.fm/issues/issue_2003_11/column1

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.

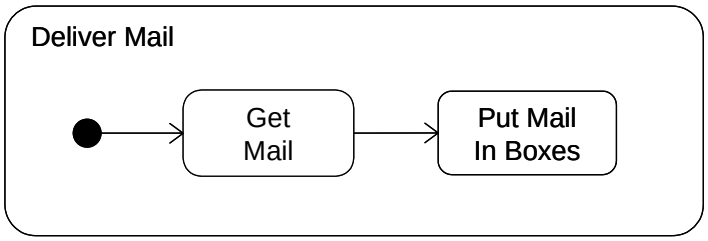


Figure 2: Initial Node

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.

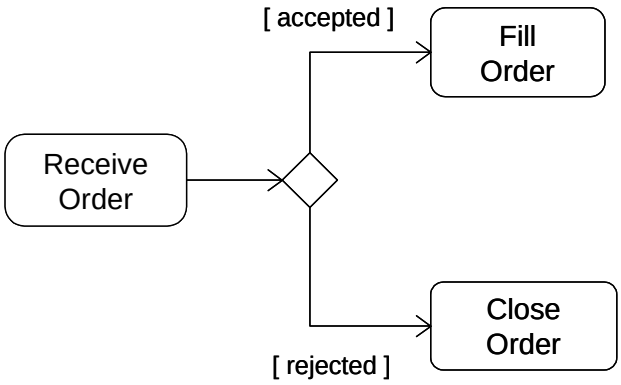


Figure 3: Decision Node

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].

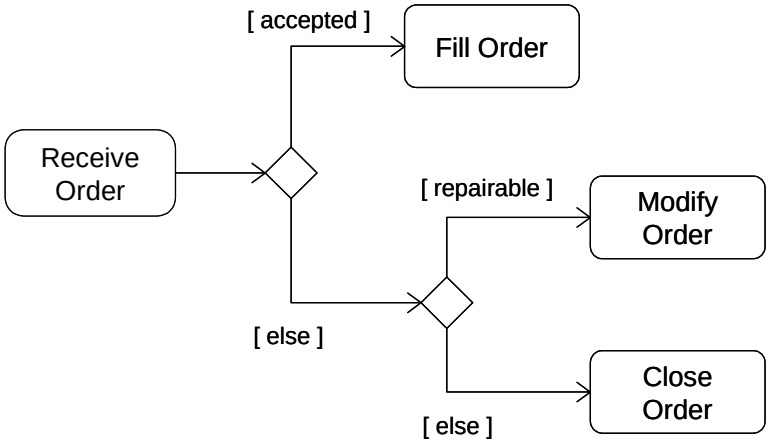


Figure 4: Chained Decision Nodes

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.

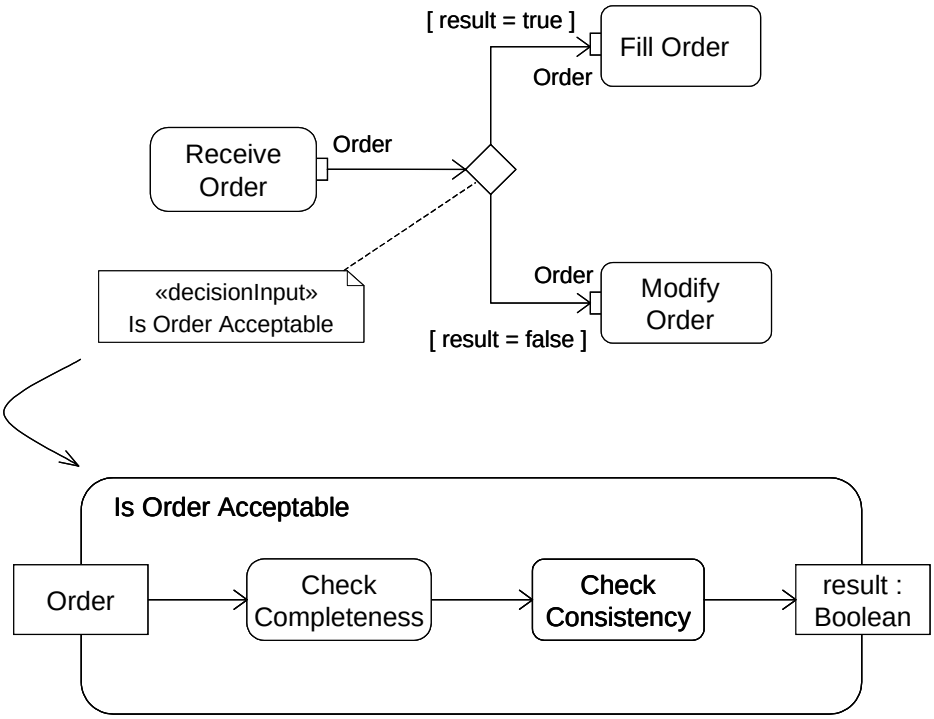


Figure 5: Decision Input Behavior

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.

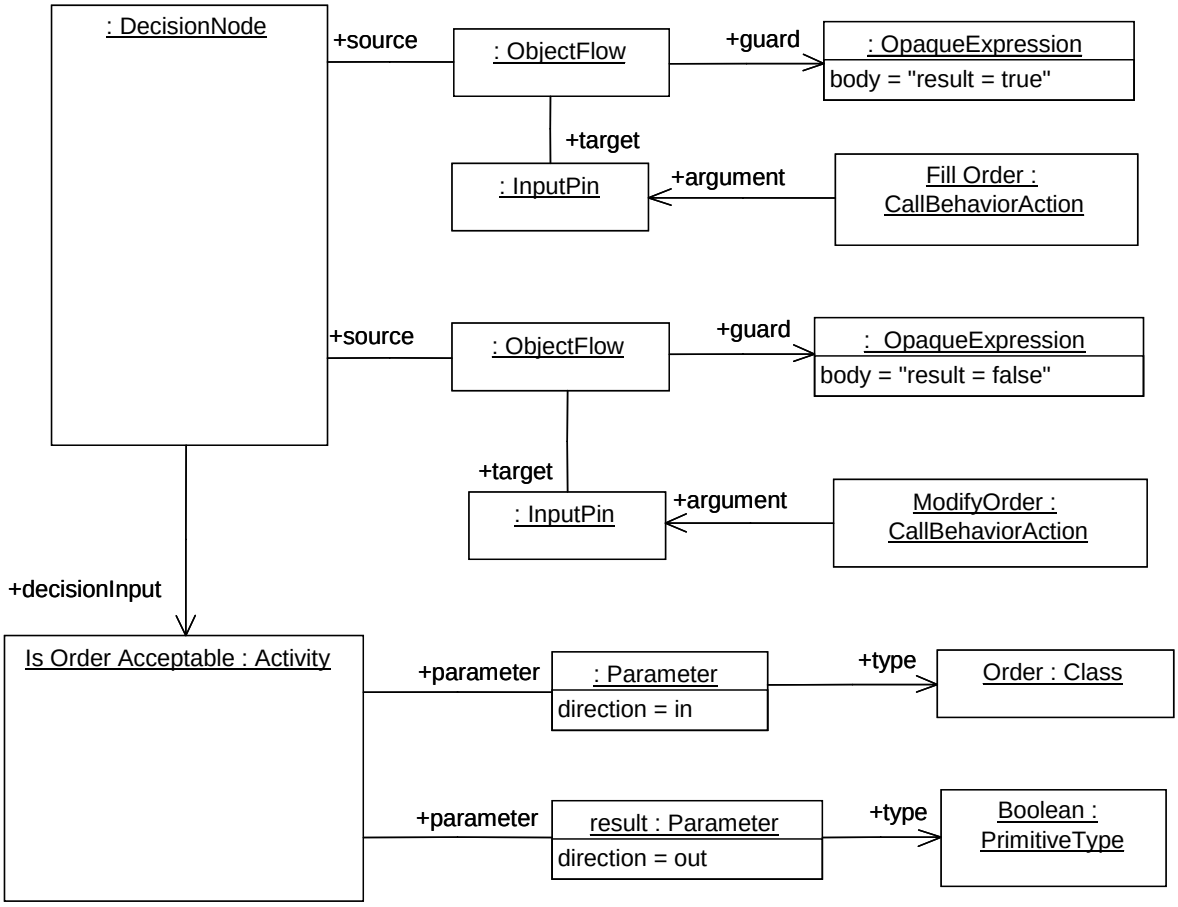


Figure 6: Repository for Part of Figure 5

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.

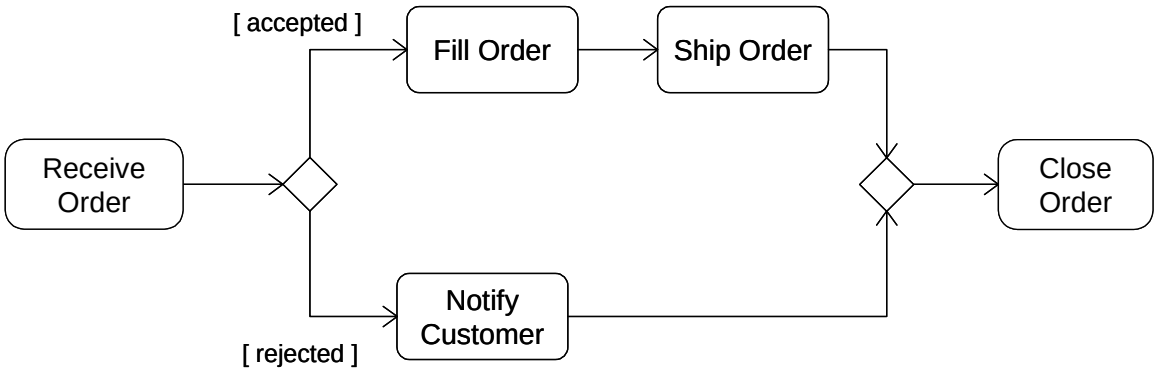


Figure 7: Merge Node with Alternate Flows

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].

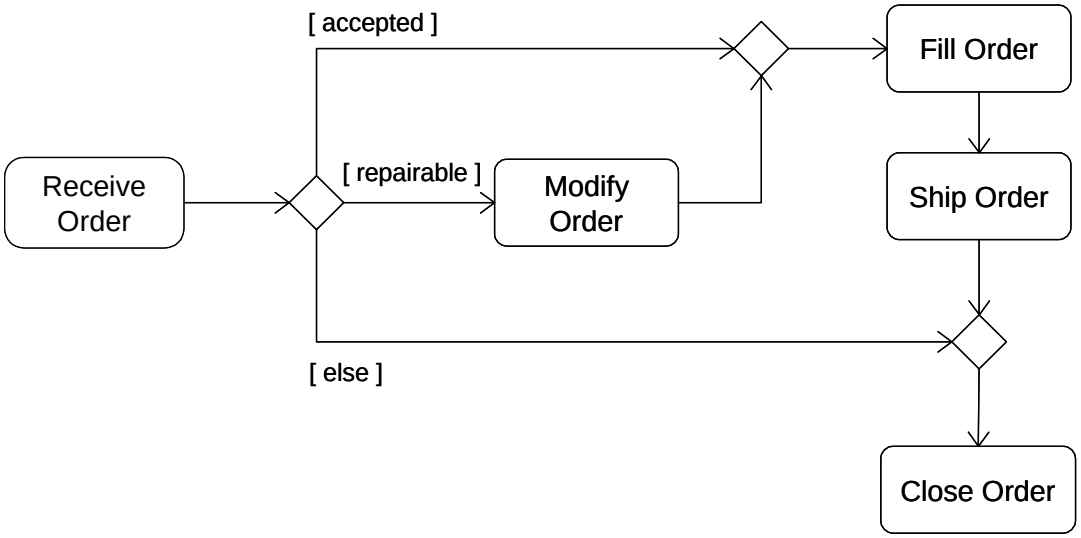


Figure 8: Merge Nodes without Nesting

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.

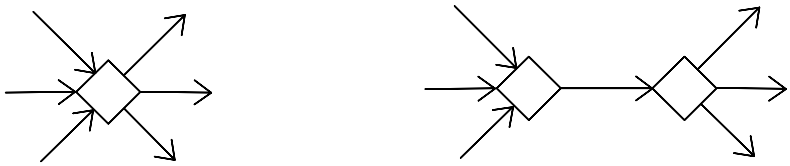
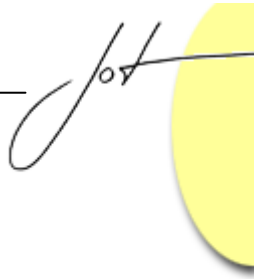


Figure 9: Merge/Decision Combination



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].

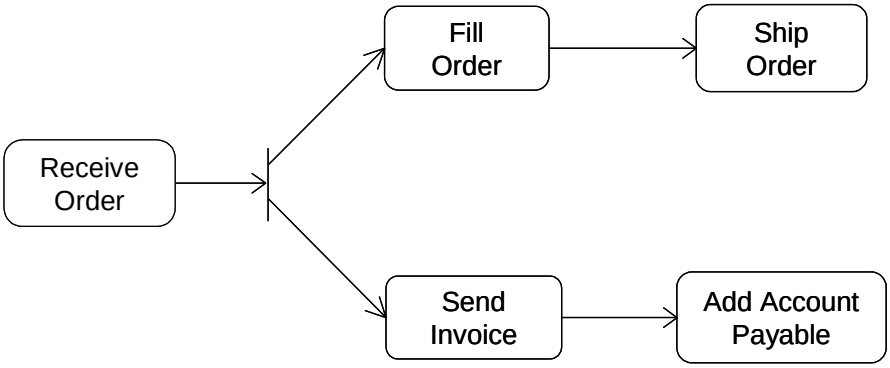


Figure 10: Fork Node

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.

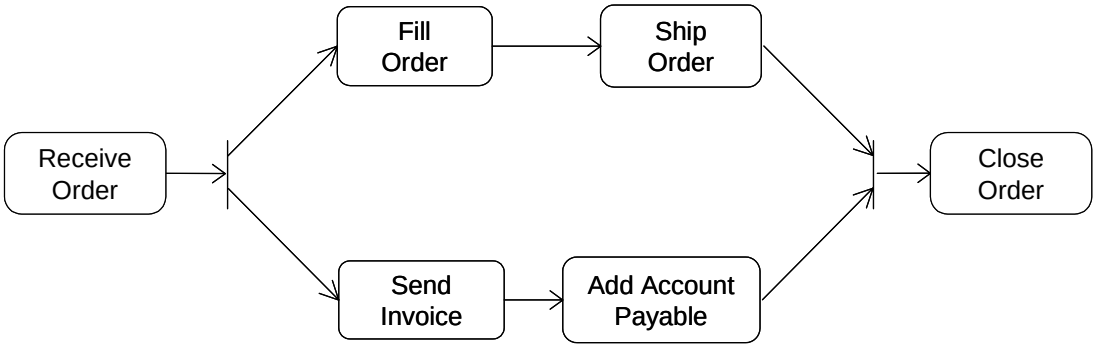


Figure 11: Join Node

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.

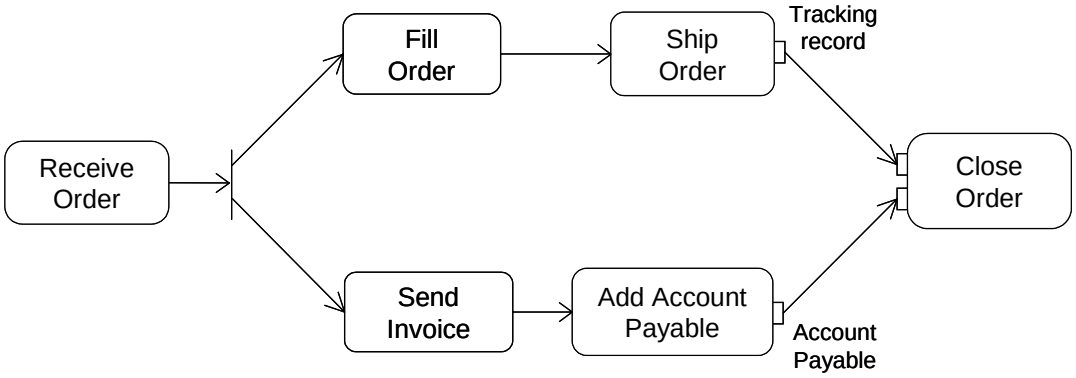


Figure 12: Joining Data Flows with Pins

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].

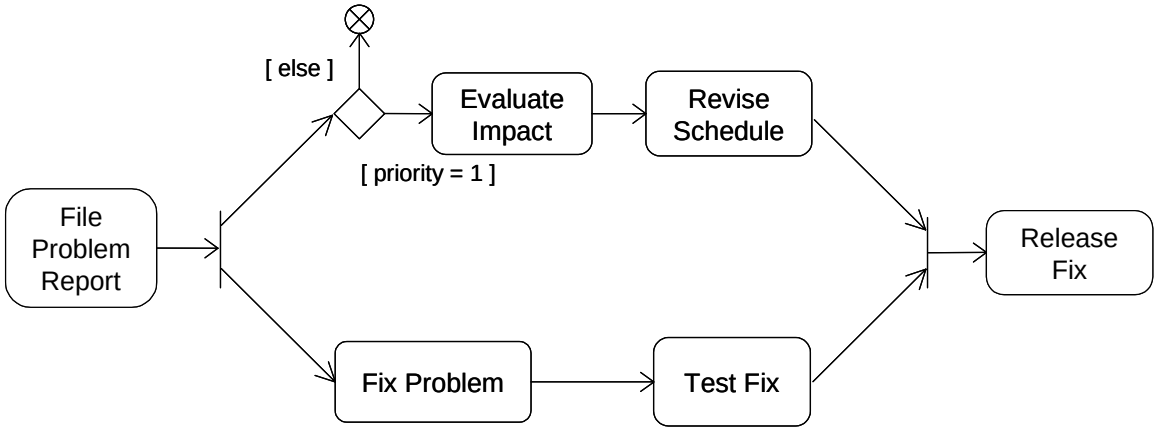


Figure 13: Join Node Anti-pattern

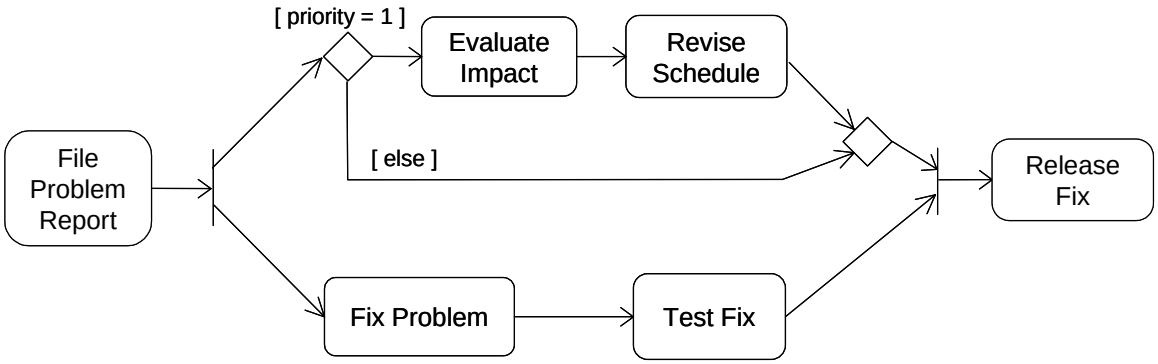


Figure 14: Join Node Pattern

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.

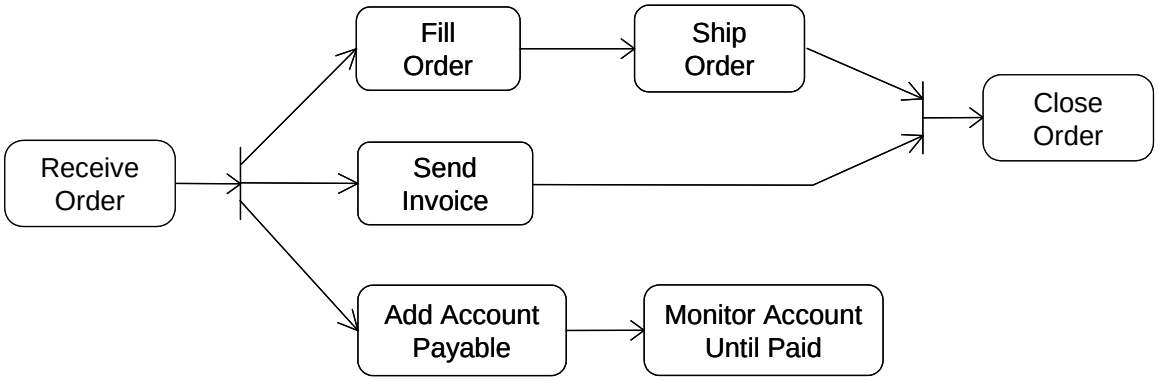


Figure 15: Fork with Partial Join

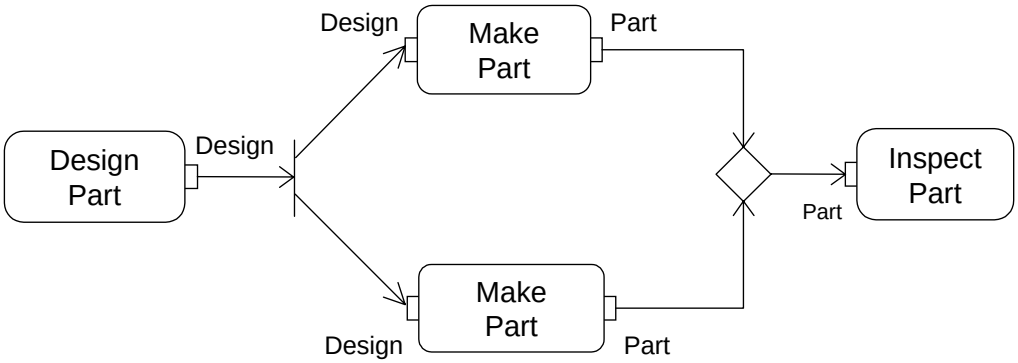


Figure 16: Fork with Merge

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.

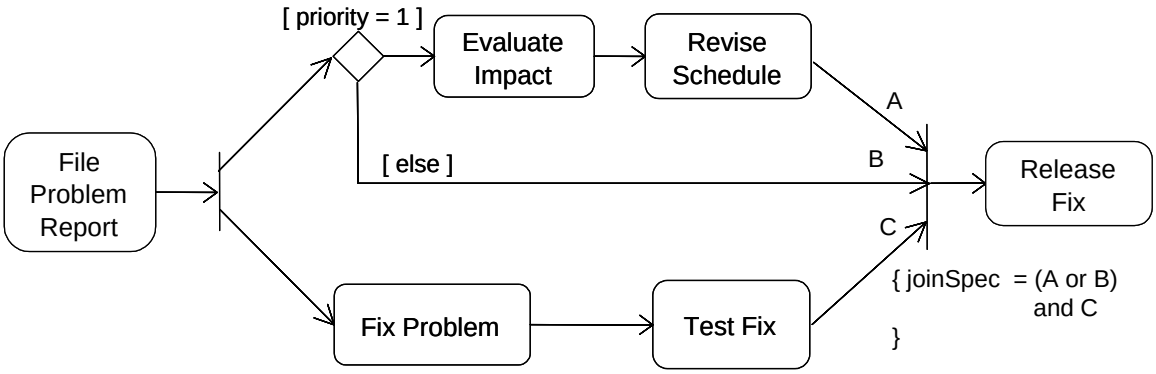


Figure 17: Join Specification

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.

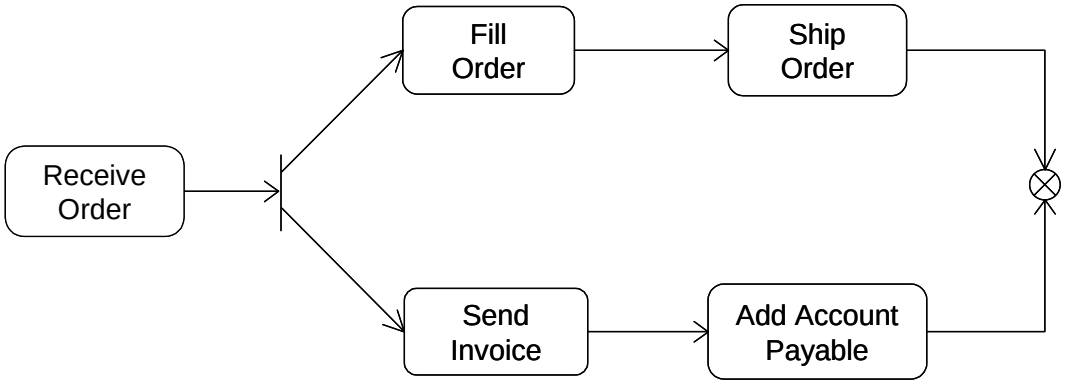


Figure 19: Flow Final Node

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.

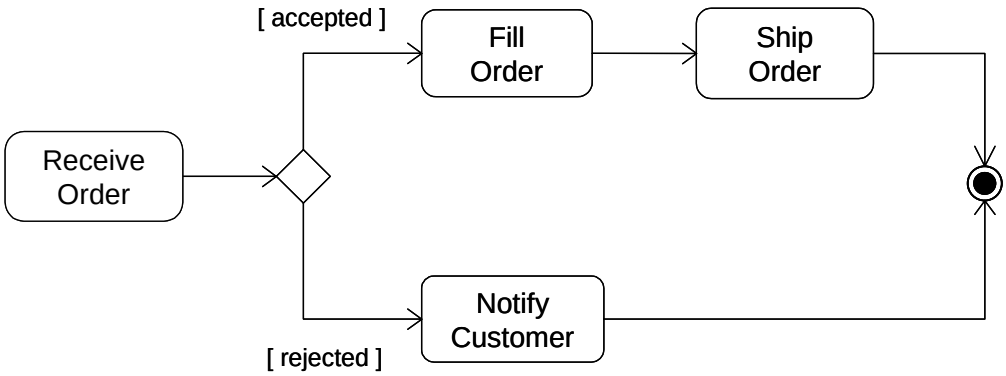


Figure 20: Activity Final Node

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.

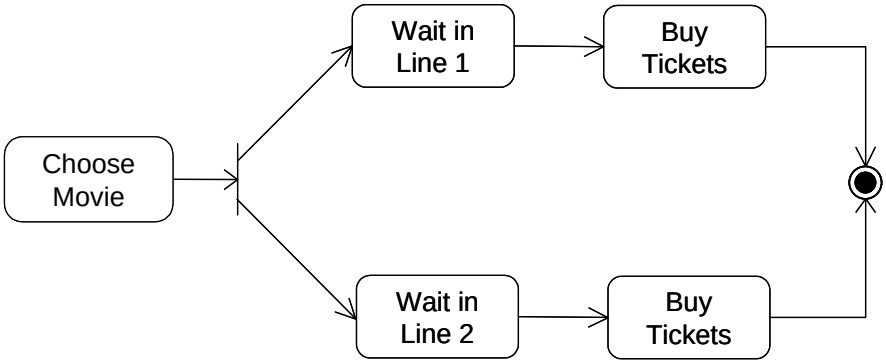


Figure 21: Activity Final Node, Racing Example

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.