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.
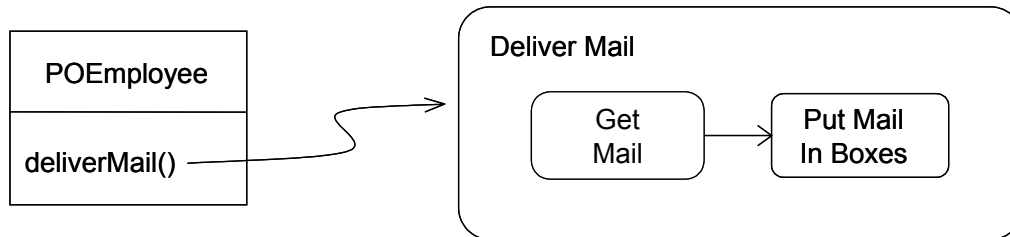
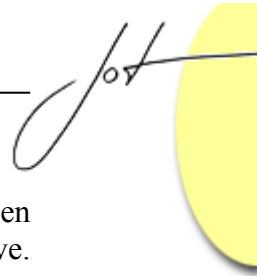Figure 1: Operation Using a Behavior as a Method

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.
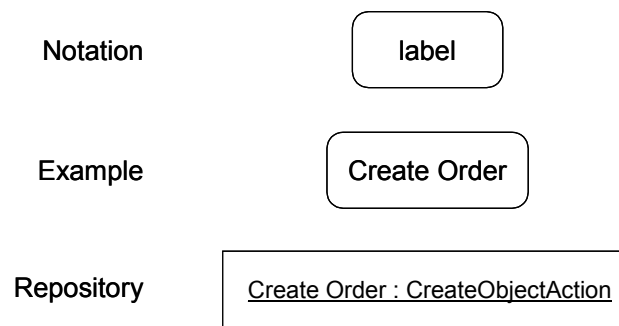
Notation   [label]

Example   [Create Order]

Repository | Create Order : CreateObjectAction |

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.
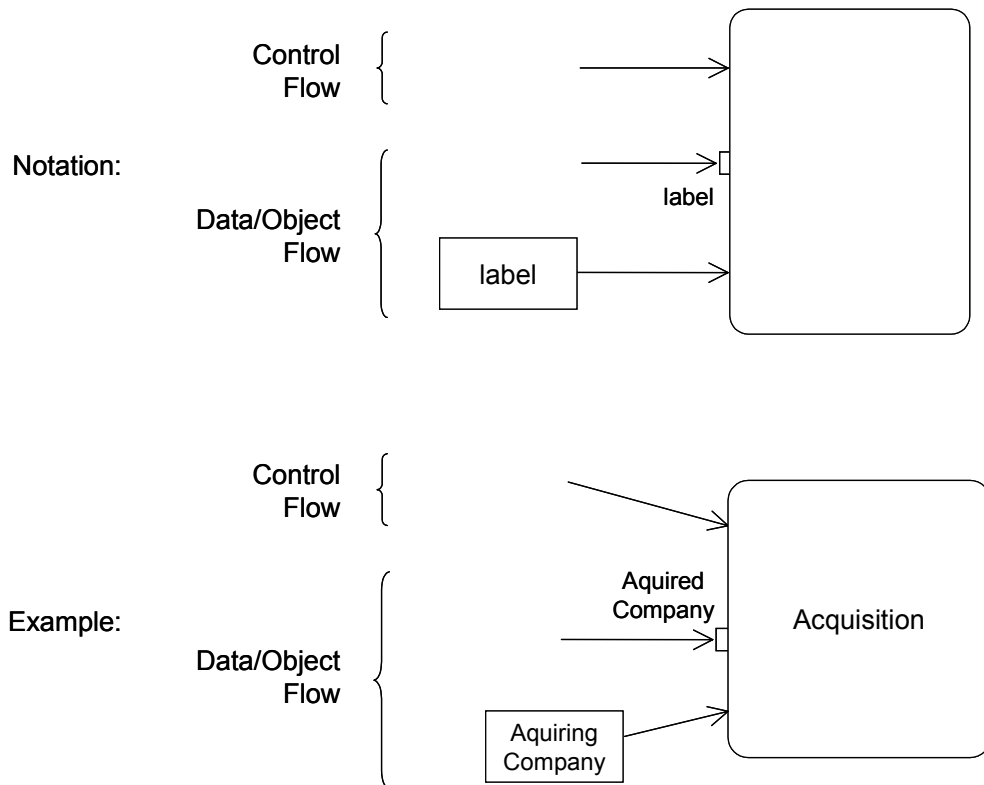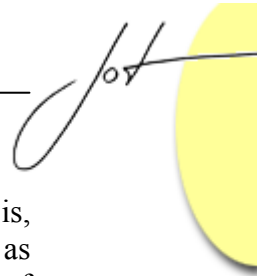

Figure 3: Control and Data Flow into an Action

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].
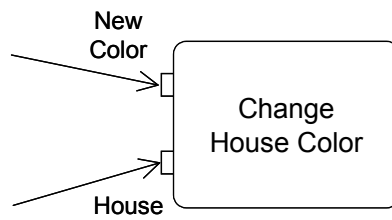
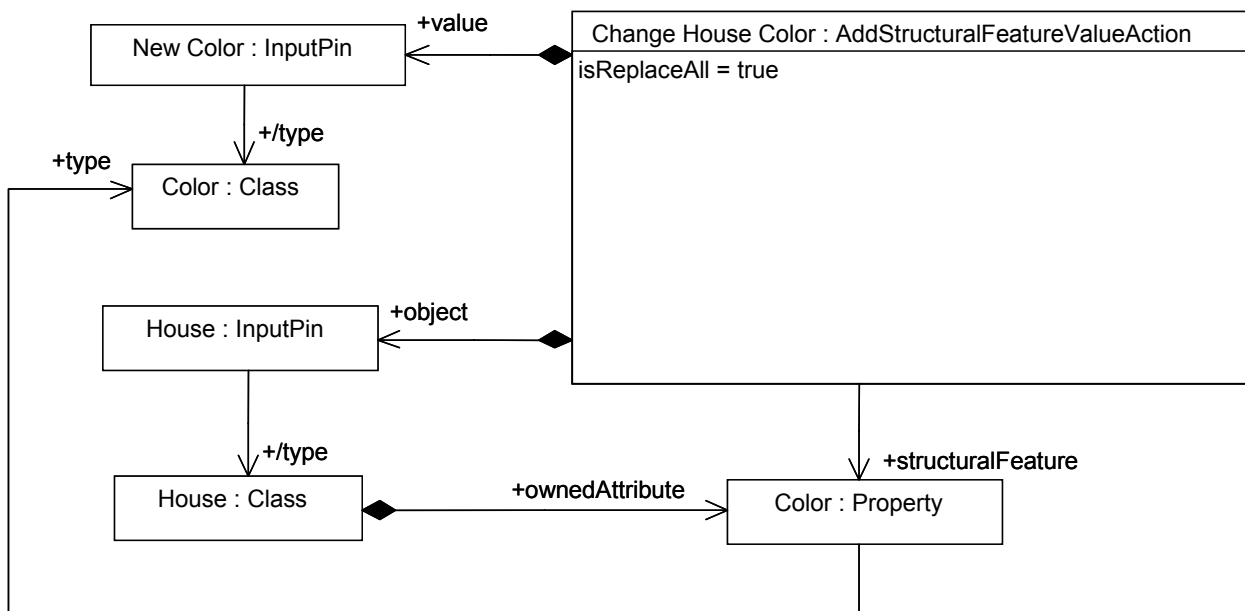Figure 4: Action for Setting an Attribute Value

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.
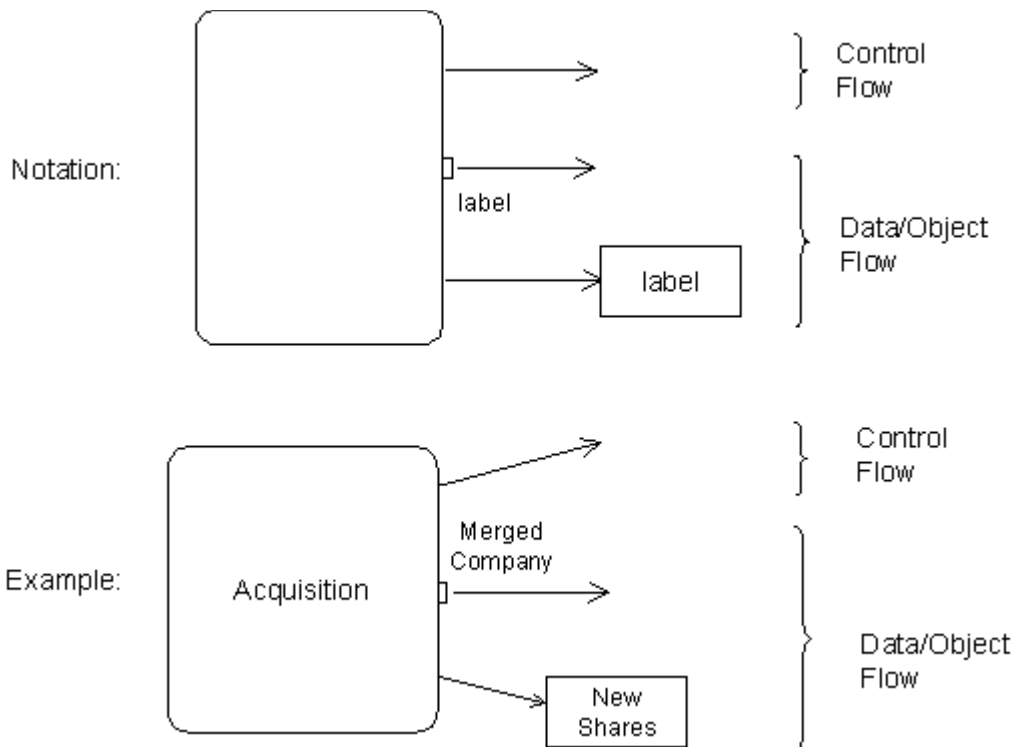
Figure 6: Control and Data Flow out of an Action

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 <u>CALLBEHAVIORACTION : ACTION</u> is linked directly to <u>DELIVER MAIL BY TRUCK : BEHAVIOR</u>, rather than going through an operation.
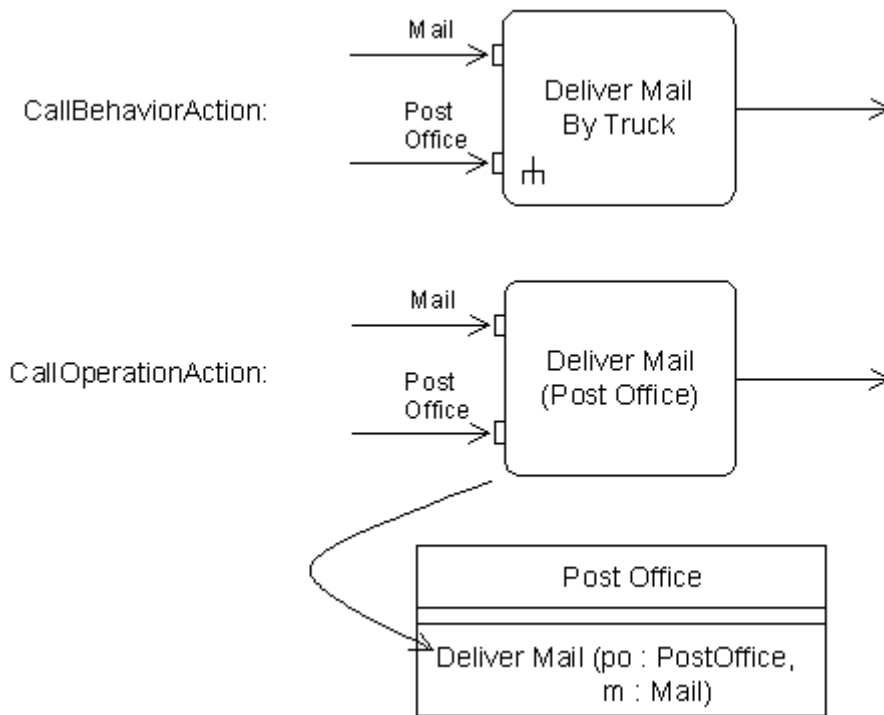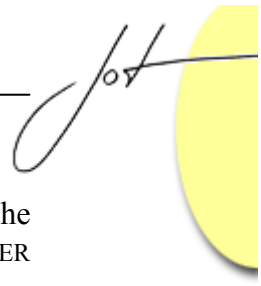

Figure 7: Examples of CALLBEHAVIORACTION and CALLOPERATIONACTION

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".
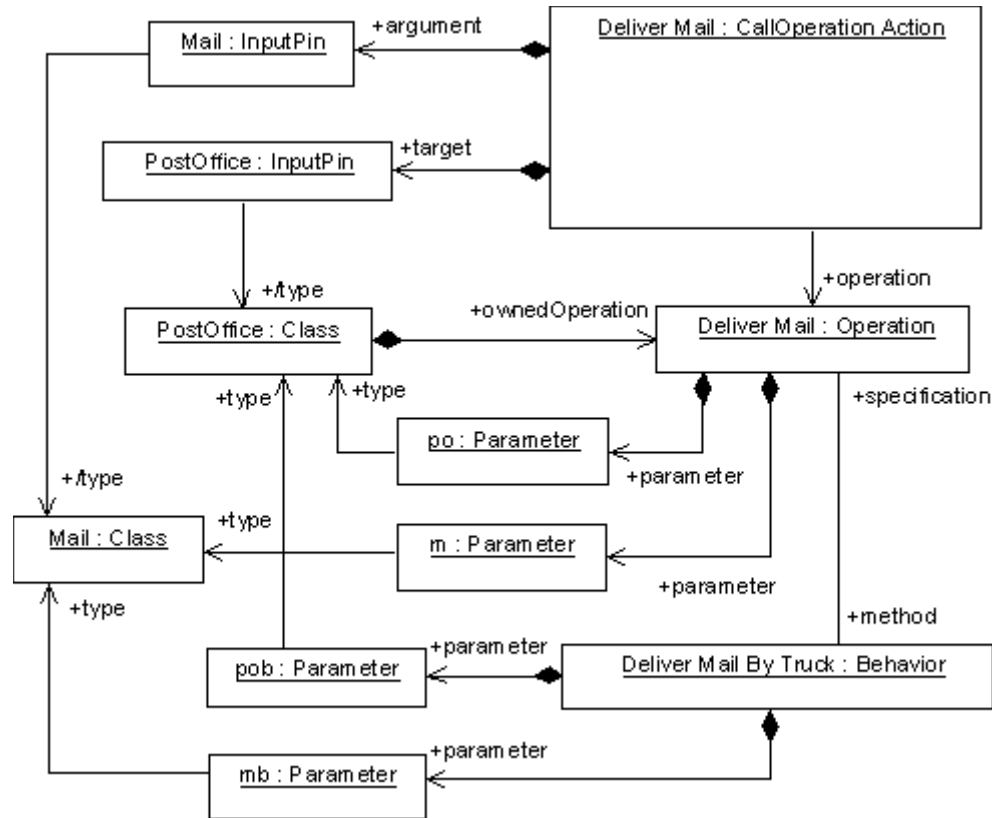

Figure 8: Repository Model for CALLOPERATIONACTION in Figure 7.

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.
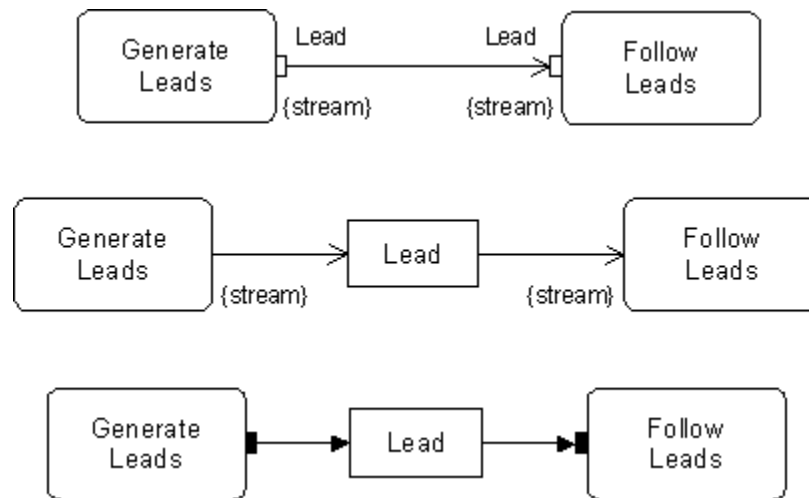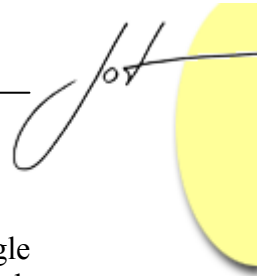

Figure 9: Streaming Parameters

---

[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]
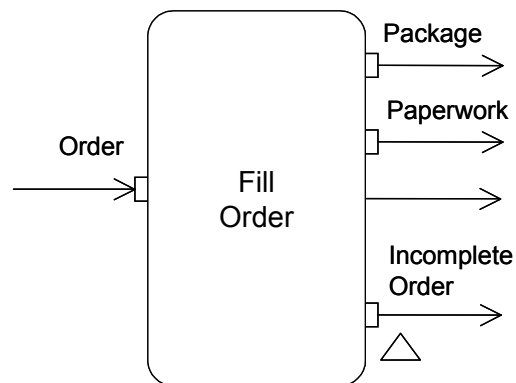

Figure 10: Exception Output Parameter

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.
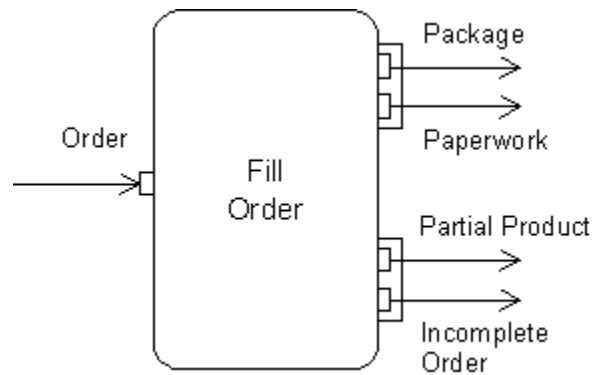

Figure 11: Parameter Sets

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.
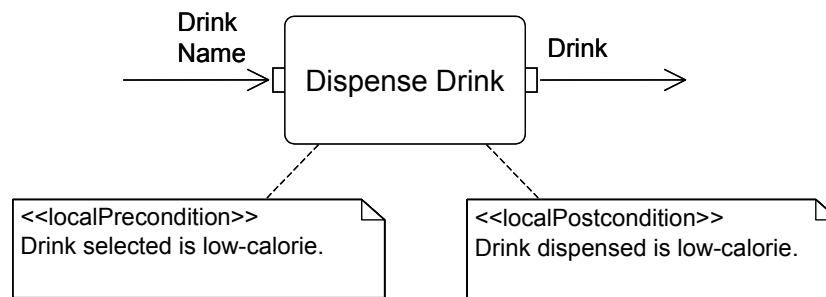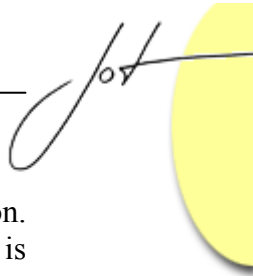

Figure 12: Local Preconditions and Postconditions

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

---

[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 .