Regular Paper

SysML and UML 2 Support for Activity Modeling*

Conrad Bock†

U.S. *National Institute of Standards and Technology*, 100 *Bureau Drive*, *Stop* 8263, *Gaithersburg*, MD 20899-8263

Received 29 July 2005; Revised 1 November 2005; Accepted 4 November 2005, after one or more revisions
Published online in Wiley InterScience (www.interscience.wiley.com).
DOI 10.1002/sys.20046

ABSTRACT

This article describes activity modeling as specified by the Systems Modeling Language (as specified by the SysML Merge Team, http://doc.omg.org/ad/2006-02-01, February 2006) and the finalization of the Unified Modeling Language version 2 (UML 2). It reviews and updates an earlier proposed alignment between Enhanced Functional Flow Block Diagrams (EFFBD), UML 2 Activities, and requirements developed by the International Council on Systems Engineering and Object Management Group. It presents a spectrum of activity modeling techniques, ranging from a widely used systems engineering diagram, the EFFBD, to continuous flow modeling. The techniques include control capabilities, continuous system concepts, and others related to functional decomposition and allocation. The article also describes refinements of activity modeling concepts identified during SysML development. © 2006 Wiley Periodicals, Inc. Syst Eng 9:160–186, 2006

Key words: systems modeling; UML 2 Activities; EFFBD; functional flow; continuous systems

1. INTRODUCTION

Recognizing the need for a standard systems engineering (SE) modeling language, the International Council on Systems Engineering (INCOSE) initiated an effort with the Object Management Group (OMG) to extend the Unified Modeling Language version 2 (UML 2) for full-lifecycle systems engineering [Friedenthal and Burkhart, 2003; SE-DSIG, 2002, 2005].[1] Requirements were developed for a UML-based language suitable for the analysis, specification, design, and verification of a wide range of complex systems (UML SE RFP) and

*This article is a US Government work and, as such, is in the public domain in the United States of America. 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.

† E-mail: conrad.bock@nist.gov

[1] An earlier article reviewed the development of this initiative, the applicability of UML to systems engineering, and related work [Bock, 2003a].

issued through the OMG [SE-DSIG, 2003]. They call for a comprehensive, consistent, and standards-based representation of systems across the development lifecycle. A priority was set on aligning the underlying meaning of the extended UML with traditional systems engineering models, and with the International Organization for Standardization's emerging Application Protocol 233 for Systems Engineering (AP-233), within ISO 10303, informally known as the Standard for the Exchange of Product model data (STEP). This will ensure that tools implementing the UML extension can reliably interchange system specification, analysis, and design with other systems engineering tools.

Industry and government responded with the Systems Modeling Language (SysML) extension to UML 2 [SysML Merge Team, 2006; Bock, 2005a; Friedenthal and Kobryn, 2004], and with updates to UML 2 during its finalization [OMG, 2004, 2005]. As input to this work, an earlier article [Bock, 2003a] gave a detailed comparison of UML 2 Activities with the Enhanced Functional Flow Block Diagram (EFFBD) [Long, 2002; Long et al., 1975; Skipper, 2005; Blanchard and Fabrycky, 1990; Grady, 1993; Kockler et al., 1990; Oliver, Kelliher, and Keegan, 1997], and with the INCOSE and OMG requirements for activity modeling in the UML SE RFP. It described how UML 2 Activities supported or did not support EFFBD and UML SE RFP requirements at the time, and suggested solutions for consideration in aligning them. This article assumes the reader is familiar with the earlier article, since it explains the correspondence between UML 2 Activities, EFFBD, and UML SE RFP requirements, most of which still holds. A brief review is given in Section 4.1.

To facilitate the understanding and application of SysML and UML 2 Activities, Section 2 organizes activity modeling features along a spectrum of possible usage patterns, ranging from activities that accept inputs and provide outputs only when they start and finish (*nonstreaming*), to activities that accept inputs and provide outputs anytime during their operation (*streaming*). To accurately describe the details of this spectrum, Section 3 refines activity concepts, which are used in the rest of the article. Section 4 addresses the nonstreaming end of the spectrum with an updated translation between EFFBD and UML 2 Activities. Section 5 addresses the streaming end with a description of SysML and UML 2 features that support it. It also refines continuous system concepts and covers SysML and UML 2 support for them. Section 6 covers activity decomposition and allocation in SysML and UML 2. Section 7 identifies UML SE RFP requirements that remain to be addressed. References to UML SE RFP requirements are given by their section numbers in that document [SE-DSIG, 2003], beginning with "6.5".

The author is not aware of other work that compares the results of the SysML and UML 2 finalization with SE functional flow, or organizes the concepts into an integrated framework. However, the spectrum presented here is influenced by the description of functional interaction in Herzog and Torne [2000].

2. SPECTRUM OF ACTIVITY MODELING APPLICATIONS

The activity models of SysML and UML 2 are flexible enough for a wide range of applications. This has the advantages of any well-stocked toolbox, but also the difficulty of understanding and choosing among a variety of alternatives. To apply these features in an organized way, it is useful to arrange them on a spectrum, with common application styles on each end, and hybrids in between.

The primary characteristic distinguishing one end of the application spectrum from the other is whether activities accept inputs and provide outputs while they are executing, as illustrated in Figure 1. At one end of the spectrum, activities accept inputs only when they start, and provide outputs only after they finish. For example, an addition function accepts two numbers, adds them, and produces a result, with no inputs or outputs while it is adding. This article refers to this end of the spectrum as *nonstreaming* activities. Items do not flow in and out of nonstreaming activities while they are executing, except perhaps for consuming resources. At the opposite end are applications in which activities pass items between each other anytime while they are executing. For example, subsystems often interact with others during their operation, such as the engine in a car, which delivers power to the clutch as it runs. The dynamics of these are modeled with *streaming* activities. In UML, the ends of the spectrum are distinguished by whether activities have streaming parameters, which are parameters that accept input or provide output while the activity is executing. Nonstreaming activities have no streaming parameters. Streaming activities have at least one streaming parameter.[2]

A secondary feature distinguishing the ends of the spectrum is whether activities terminate themselves, or are terminated by other activities. At the nonstreaming end of the spectrum, activities terminate when their own internal logic determines the task is done. They accept inputs at the beginning of execution, operate on them,

[2]SysML defines nonnormative stereotypes «streaming» and «nonStreaming» for activities that apply these constraints.

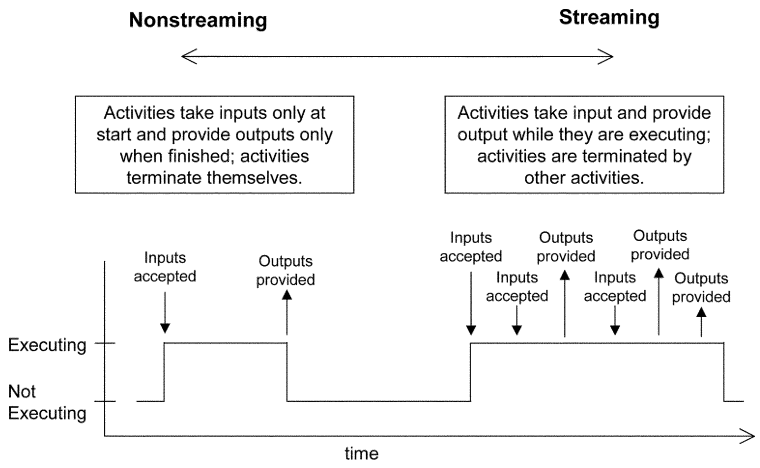


Figure 1. Spectrum of activity modeling applications.

and provide outputs when the activity is complete. At the streaming end of the spectrum, inputs can be accepted and outputs produced by an activity that is already executing, so an activity can potentially operate indefinitely. They often require specialized external control activities to determine when they should terminate. Streaming activities start with whatever inputs are available, continue to accept inputs and provide outputs for an indefinite period, until a control activity terminates them.[3]

Choosing one end of the spectrum can simplify modeling by reducing the choices available for other capabilities in SysML and UML 2 Activities. In particular:

- Control values

In nonstreaming applications, where activities terminate themselves, it is not necessary to stop them by some external action. They just run until they produce outputs. This is why EFFBD functions only enable other functions to start, rather than disable them once they are executing. There is one control value, for enabling.

In streaming applications, where activities run until disabled by other activities, control must be extended to support turning them off; otherwise they might run forever or at times they should not. There are at least two control values in streaming activities, enabling and disabling. Activities that produce control values are called *control operators*. See Section 5.1.1 for more detail.

- Buffering

In nonstreaming applications, where activities do not accept input while they are executing, it is useful to queue inputs until the activity is ready to accept them, as in EFFBD.

In nonstreaming streaming applications, where activities accept input while they are executing, inputs can flow into the activity at any time, and the activity can maintain its own buffers if necessary. Buffering is only needed when activities are not executing. See Section 5.1.2 for more detail.

- Required and optional inputs

In nonstreaming applications, where activities can accept inputs only when they start, some activities might need to proceed without all inputs available. Inputs arriving late are accepted in later executions of the activity. This is why EFFBD provides both nontriggering and triggering inputs, which are equivalent to lower multiplicities on parameters in UML (see Section 4).

In streaming applications, where activities can accept inputs while they are executing, an optional input means it may never arrive at all, which is not a useful input. Normally, inputs to streaming activities are required. Since streaming inputs can arrive anytime while the activity is executing, a required input only means it must arrive at some point during each execution.[4]

[3] In the functional interaction form of behavior described in Herzog and Torne [2000], the default function runs for the life of the system, taking inputs and outputs at any time. This is specialized by using one of two kinds of control port. The first kind can only enable the function, which executes until it is finished, as in EFFBD. The second kind of control port can enable or disable the function, a capability placed at the streaming end of the spectrum in this paper.

[4] A more complete model would specify whether required and optional inputs apply to starting the function or to completing it, and support this independently of whether the function can take input while it is executing. Currently UML and SysML support required/optional to start for non-streaming parameters, and required/optional to complete for streaming parameters.

- Multi-exit/entry

In nonstreaming applications, where activities do not provide output while they are executing, it is useful to provide items along different output flows at each execution of the activity. These are EFFBD multi-exit functions, or parameter sets in UML (see Section 4).

In streaming applications, where activities provide outputs while they are executing, and potentially execute for an indefinite period depending on control operators, it is not commonly useful to have some outputs be exclusive of other outputs for the entire duration of the activity. Especially over a long period of time, an activity may provide items at any or all of the outputs at some point.

- Concurrent execution

In nonstreaming applications, where activities do not provide output while they are executing, it is possible for inputs to arrive at a faster rate than the activity can operate on them, causing backup in the flow. Activities that support concurrent executions can operate on inputs in parallel to clear backups (see Section 5.2.2).

In streaming applications, where activities accept inputs while they are executing, and in the extreme case of activities that operate for the life of the system, concurrent execution is not necessary, because there is only one execution of each activity that accepts all inputs for that activity however quickly they arrive. Activities keep their own internal buffers and manage concurrency as necessary. They can also declare the rates of flow they are able to handle (see bullet item below and Section 5.2.2).

Some applications are in the middle of the spectrum, using aspects of both nonstreaming and streaming activities. These necessarily have more choices, and consequently are more complicated to design. One simplification is to specify each activity with one end of the spectrum or the other, rather than mixing features from each end within a single activity. For example, define activities where the inputs and outputs are either all streaming or all nonstreaming, rather than activities where some inputs and outputs are streaming and others are not. The two kinds of activity can be used together in the same enclosing activity, even though the enclosing activity is from one end of the spectrum or the other. See the example in Figure 26 of Section 5.2.2. Sometimes it may be necessary to have some activities take features from both ends of the spectrum, with both streaming and nonstreaming parameters on the same activity. For example, a manufacturing process may assemble a set of parts into a product, but during assembly interact with many other activities. This is the most complicated kind of system to design, because it has very few constraints on how to combine activity capabilities. It will be addressed in future work.

Some capabilities of SysML and UML 2 Activities are useful across the spectrum of Figure 1:

- Usage and definition

It is economical in many applications to define an activity once and use it many times in defining other activities. This has the advantage that any changes to the reused activity are effective for all the other activities that use it. See Section 3.

- Rate of flow

It is useful to model the rate at which items and input items arrive at an activity or the rate at which output items leave it, especially for flows that might be subject to backup and bottleneck. Even with streaming activities, there may be rates of input and output they are not able to support. This can be declared on activities to avoid application in situations they cannot support. See Section 5.2.2.

Table I. Capability Choices along Activity Spectrum

Activity Feature	Nonstreaming Activities	Hybrid Activities	Streaming Activities
Input and output during execution	No	Can vary by activity, or within a single activity.	Yes
Control values	Enabling only		Yes
Buffering	Yes		When disabled only.
Required and optional inputs	Yes		Required only
Multi-entry/exit	Multi-entry		No
Concurrent executions	Yes		No
Usage and definition	Yes		
Rate of flow			
Variation in item value			

- Variation in input or output item value

When input or output items are data values, such as numbers, or are objects that have data value properties, these values will vary over time as new input items arrive at an activity or output items leave it. The variation might be described by a mathematical function, which can produce values drawn from a finite or infinite set. Value variation is applicable to activities that are expected to have multiple values flow through at one time, which applies across the spectrum. See Section 5.2.

Table I summarizes the above capabilities available on the spectrum. The next section refines common activity concepts before elaborating on the nonstreaming end of the spectrum in Section 4, and streaming activities in Section 5.

3. ACTIVITY CONCEPT REFINEMENTS

To better understand the details of the spectrum given in Section 2, the SysML development team found it necessary to refine commonly used activity concepts. These refinements are not immediately evident in graphical diagrams, because diagrams compress multiple concepts into a small number of graphical constructs.[5] This has the benefit of readability, but hampers the understanding, implementing, and applying the language. To address these problems without impairing usability, the multiple underlying concepts and their relations must be identified, and tied to the graphical notation. Section 3.1 describes the most general concepts, and Sections 3.2 and 3.3 refine them.

3.1. Item, Activity, and Control

The most basic activity concepts correspond to the intuitive notions of "things that change," "how they change," and "when they change":

Item: the most general notion of an entity that may flow through a system, whether physical or informational. It includes physical matter and objects, energy, data, and software objects.

Activity: the transformation of items by taking items as inputs and providing items as outputs.

Control: the determination of when activities perform their transformation. This includes starting as well as stopping a transformation. Activities must obey control they receive, whereas they have leeway in how to handle items, see Section 5.1.1.

SysML further identifies basic concepts for activities and items as follows:

Definition: the description of an activity or item independent of how it is used by other activities.

Usage: how an activity or item is used in the context of activities.

Instance: the actual execution of an activity, or particular item as it flows during execution.

The next two sections apply the above refinements to activities and items, and show how they are rendered in compact graphical notations.

3.2. Activity Definition, Usage, and Execution Instance

Activity diagrams compress multiple activity concepts into a single kind of graphical shape, a round-cornered rectangle in UML 2, or a rectangle in EFFBD. For example, Figure 2 is a UML activity diagram showing the same activity, HEAT LIQUID, used more than once (item flow is omitted for brevity). It has a single definition, shown in Figure 3, but each usage in Figure 2 has different control flow lines coming into and going out. One has a flow coming from FILTER LIQUID and going to WASH INSTRUMENTS, the other has a flow coming from a join (concurrency in EFFBD) and going to WASH HANDS. Activity usages (actions in UML) are needed to specify which flow lines applies to which usage of HEAT LIQUID. This way, the underlying model for the diagram has a way of identifying each usage separately from the single definition, even though they are displayed with the same shape and usually have the same label. For clarity, the label can include the usage name in the SysML extension to UML notation (see Section 6.1).

The activities in Figures 2 and 3 may be executed many times during the life of the system, which means

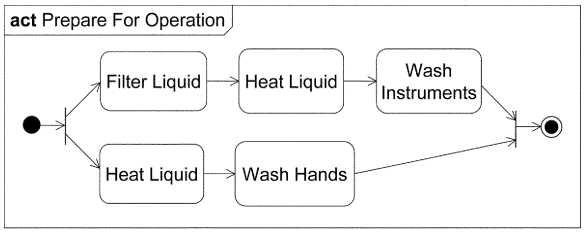


Figure 2. Multiple usages of the same activity.

[5] It is said that a picture is worth a thousand words, but it is not always clear which thousand. The distinction between language-as-pictures/text and language-as-concepts is reflected in OMG's Model-driven Architecture as the difference between notation and metamodel [Bock, 2003b; OMG, 2006].

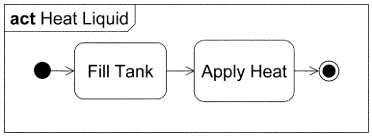


Figure 3. Activity definition.

each activity usage may represent many individual executions of an activity. For example, Filter Liquid is only used once, but if PREPARE FOR OPERATION is executed many times, then FILTER LIQUID will be also. This is important for accurately describing the execution traces implied by the flow model, and in particular to specify SysML and UML 2 control functionality (see Sections 4.3 and 5.1.2).

The example above illustrates the concepts underlying the activity notation:

1. Activity Definition (Activity in UML)[6]: an activity defined independently of how it is used in any diagram. For example, an activity for heating a liquid only specifies that the temperature of the output liquid will be higher than the input, but not where the liquid comes from or where it goes to, or exactly what kind of liquid is heated.
2. Activity Usage[7] (Action in UML)[8]: how an activity is used in a definition of another enclosing activity. For example, a usage of the activity for heating liquids is needed to specify which other activities provide the liquid to be heated and which receive it after heating. Activities are used in the definitions of other higher-level activities, which are the context of the usage (see Section 6).
3. Activity Instance or Execution (same in UML): an individual performance of an activity, with particular start and stop times, and operating on particular items. An activity may execute many times over the life of a system, receiving items from and providing items to other activity executions. Each time an activity is enabled, it is a new activity execution, even if the usage is the same. For example, an execution of a heating activity will occur between particular start and stop times, operating on a particular inflow of liquid from another executing activity, and providing an outflow of liquid to another executing activity. The particular items operated on by an activity execution are item instances (see Section 3.3).

[6]Activities are one of three kinds of behavior model in UML, the other two being interactions and state machines. Activities highlight how outputs of one subfunction flow to the inputs of another, while interactions focus on messages between objects, and state machines emphasize object states and transitions between them based on incoming signals. SysML models function as UML 2 activities because they are the closest UML behavior to SE functional flow diagrams.

[7]The functional interaction form of behavior described in Herzog and Torne [2000] uses the term "function instance" instead of "function usage."

[8]UML has many kinds of action. Function usages in the SE sense correspond to CallBehaviorAction.

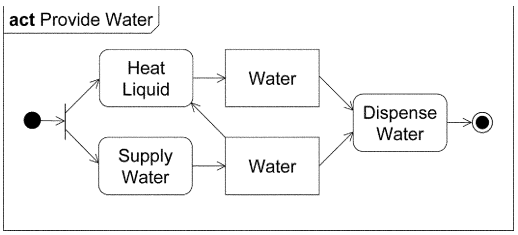


Figure 4. Multiple items of same kind input to an activity usage.

3.3. Item Definition, Item Usage, Item Instance

Activity diagrams compress multiple item concepts into a single kind of graphical shape, a rectangle in UML 2, or a round-cornered rectangle in EFFBD. For example, Figure 4 shows an activity with the same item definition (classifier in UML[9]), WATER, used as output from different activities, HEAT LIQUID and SUPPLY WATER, and input to the same activity, DISPENSE WATER. The item definition, WATER, exists independently of whether it is used in any particular activity, and is defined on a separate kind of diagram (class diagrams in UML, block definition diagrams in SysML). It would continue to exist even if PROVIDE WATER removed from the system specification.

To dispense water of the proper temperature, DISPENSE WATER must be able to tell the inputs apart. This requires item usages on the activity definition (parameters in UML), as shown in Figure 5, omitting the internal flows for brevity. Parameters have a similar notation to the items in Figure 4 because they both refer to items flowing in and out of activities. Parameters are named to distinguish each input, shown the left of the colon, with item definition or type shown the right of the colon (parameter type in UML).

Once item usages (parameters) are specified on activity definitions, the flows that go in and come out of them in a particular activity usage must be determined. For example, the hot water input to the usage of DISPENSE WATER in Figure 4 comes from the usage of HEAT LIQUID, and the cold water comes from the usage of SUPPLY WATER. However, the notation of Figure 4

[9]See Footnote 12 for SysML terminology.

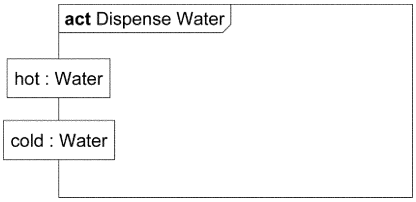


Figure 5. Multiple items of the same kind input to an activity definition.

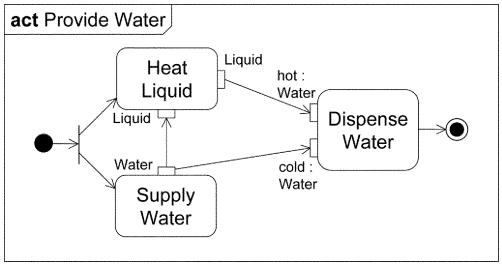


Figure 6. Item usages on activity usages (pins).

does not show this information. In addition, other usages of DISPENSE WATER might accept inputs from different activities entirely. To support this example, the underlying model must separate the item usages for each activity usage, to record which output parameter flows to which input parameter in each usage of DISPENSE WATER. This requires another layer of item usage, the item usages on activity usages.

Items usages on activity usages are pins in UML. UML provides two notations for pins. Figure 4 is the object node notation, which is the simplest, but is limited to flows where the parameter type is the same at both ends, and where it is unambiguous which flow corresponds to which parameter, neither of which are true in this example.[10] Figure 6 is the most detailed view, showing each pin individually. They are labeled with the corresponding parameter name and type.

An activity may be executed many times over the life of the system, operating on many actual items. For example, DISPENSE WATER in Figure 4 may be executed many times, dispensing many instances of WATER. Identifying each item instance is important for accurately describing the execution traces implied by the flow model, and in particular to specify SysML flow rates (see Section 5.2.2). For example, during execution of the models in Figures 4 and 6, an item instance going out of SUPPLY WATER will flow in only one direction, even though there are two flow arrows going out of the object node and pin.[11]

The example above illustrates that the refinements of definition, usage, and instance apply to items as well as activities:

1. Item Definition or Item Type (Classifier in UML)[12]: a kind of item that may be input or output from an activity, independent of whether any activity actually accepts that kind of item as input or provides it as output. For example, liquids exist whether or not any activities accept them as input or output.
2. Item Usage by Activity Definition (Parameter in UML): the kind of item input or output to an activity definition, independent of how the activity is used in any diagram. For example, a heating activity might be defined to accept a liquid as input and provide it as output, but not what kind of liquid flows into a particular usage of the activity. Parameters are named, and specify the kind of items they input or output (parameter type in UML).
3. Item Usage by Activity Usage (Pin in UML): the connection point between a flow line and a parameter at a particular activity usage. For example, an input pin to a heating activity will be the target of an incoming flow line and correspond to the liquid input parameter of the heating activity. The pin might also narrow the range of possible liquids that will be heated, for example, specifically to water. However, the type of the pin must be compatible with the type of parameter.
4. Item Instance (Instance in UML): This is an individual item that is input or output from a particular execution of an activity, for example, particular quantities of liquid flowing into a heating activity operating at a certain time. In Figure 4, SUPPLY WATER has one output parameter that split along two flows in this particular usage. During execution, each item instance going out of SUPPLY WATER will flow along only one of the flow lines, to HEAT WATER or DISPENSE WATER, but not both.[13]

[10] The explicit pin notation is also required if one end of the flow has no pin, as sometimes happens with control flow (see Fig. 14).

[11] This execution trace is appropriate for physical systems. Information systems would use a UML fork to send data along multiple flow lines at once. See Footnote 1 to Table II.

[12] UML does not distinguish things that flow through a system from things that do not. It defines the general category of Classifier covering both, but divides this into classes, which have separately identifiable instances (objects), and datatypes, which are not always separately identifiable, such as two instances of the number 3. SysML uses UML's categorization, but uses the term "block" for class.

[13] SysML and UML 2 provide ways to constrain property values of item instances by ranges, distributions, equations, and other means, as well as a model of units and dimensions for these values [Bock, 2005a, SysML Merge Team, 2006].

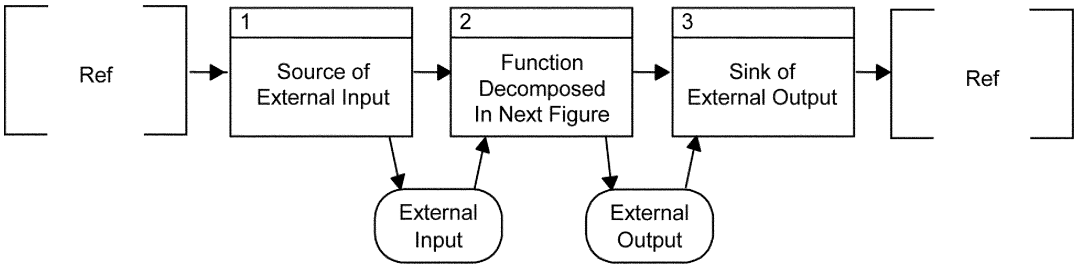


Figure 7. Enhanced functional flow block diagram context for Figure 8.

More information on the underlying models of UML activity diagrams is available in a separate series of articles [Bock, 2003c, 2003d, 2003e, 2004a, 2004c].

4. NONSTREAMING ACTIVITIES (EFFBD)

A primary example of the nonstreaming end of the spectrum in Section 2 is the Enhanced Functional Flow Block Diagram (EFFBD). It has been applied for three decades, and multiple tools have supported it over that time. There is substantial design information in existence based on this form of behavior modeling. It is a special usage of UML 2 Activities in which all activities are nonstreaming, and other constraints and usage patterns apply. To ensure accurate translation between EFFBD and SysML/UML 2 Activities, this section updates the mapping given in an earlier article [Bock, 2003a] to reflect SysML's interpretation of EFFBD, based on other standards and in consultation with EFFBD users and tool vendors [Long, 2002; Long et al., 1975; Skipper, 2005; Blanchard and Fabrycky, 1990; Grady, 1993; Kockler et al., 1990; Oliver, Kelliher, and Keegan, 1997]. Section 4.1 describes the translation between individual constructs of the languages, using one-to-one mappings and constraints on UML 2 Activity usage. Section 4.2 gives mappings between patterns of constructs in EFFBD and UML 2 and SysML. Section 4.3 gives the rules for activity execution that are consistent with EFFBD and UML 2.

Two aspects of EFFBD are not covered in the translation: replication and resources. Replication provides multiple executions of the same activity from the same activity usage (see Section 7). Resources are required to execute an activity, but are not shown as inputs. For example, electrical or computational power might be resources to a heating activity, but no flow lines are shown for them when the heating activity is used in other diagrams, even though heating will use electrical and computational power.[14] Replication and resources affect EFFBD execution, and their translation to UML 2 Activities will be addressed in future articles.

4.1. EFFBD Direct Translations to SysML and UML 2 Activities

Figures 7 and 8 show example EFFBDs illustrating the translation to UML 2 Activities with one of the elements in Figure 7 decomposed into the diagram in Figure 8 (see Section 6.1 for additional decomposition notation). Figure 9 gives the corresponding UML 2 Activity Diagram. Both EFFBD and UML activities give the sequence, inputs and outputs, and conditions for execution of activities. For example, function 2.3 in Figure 8 can only begin after function 2.1 has completed. Both types of diagram also show how the outputs of one function are passed to the inputs of others. For example, in Figure 8, function 2.2 accepts input of type ITEM 1 from the output of function 2.1. Function 2.2 cannot start until an instance of ITEM 1 arrives.

Table II shows the correspondence between constructs in the EFFBD, Activity Diagrams, and the UML SE RFP requirements, updated from a previous article [Bock, 2003a]. EFFBD is translated to activities following this table, except for the pattern translations described in Section 4.2. There are three kinds of construct:

1. Functional: for defining activities used in flow diagrams.
2. Item flow: for routing items between activities in flow diagrams.
3. Control flow: an additional way that a flow diagram determines when activities are enabled and disabled.

SysML suggests constraints on UML 2 Activities usage for its interpretation of EFFBD[15]:

1. Activities do not have partitions (see Section 6.2).
2. All decisions, merges, joins, and forks in an activity are well nested. In particular, each deci-

[14] For more discussion of views on input and output, see Bock [2004d].

[15] SysML defines a nonnormative stereotype «effbd» for activities to apply these constraints.

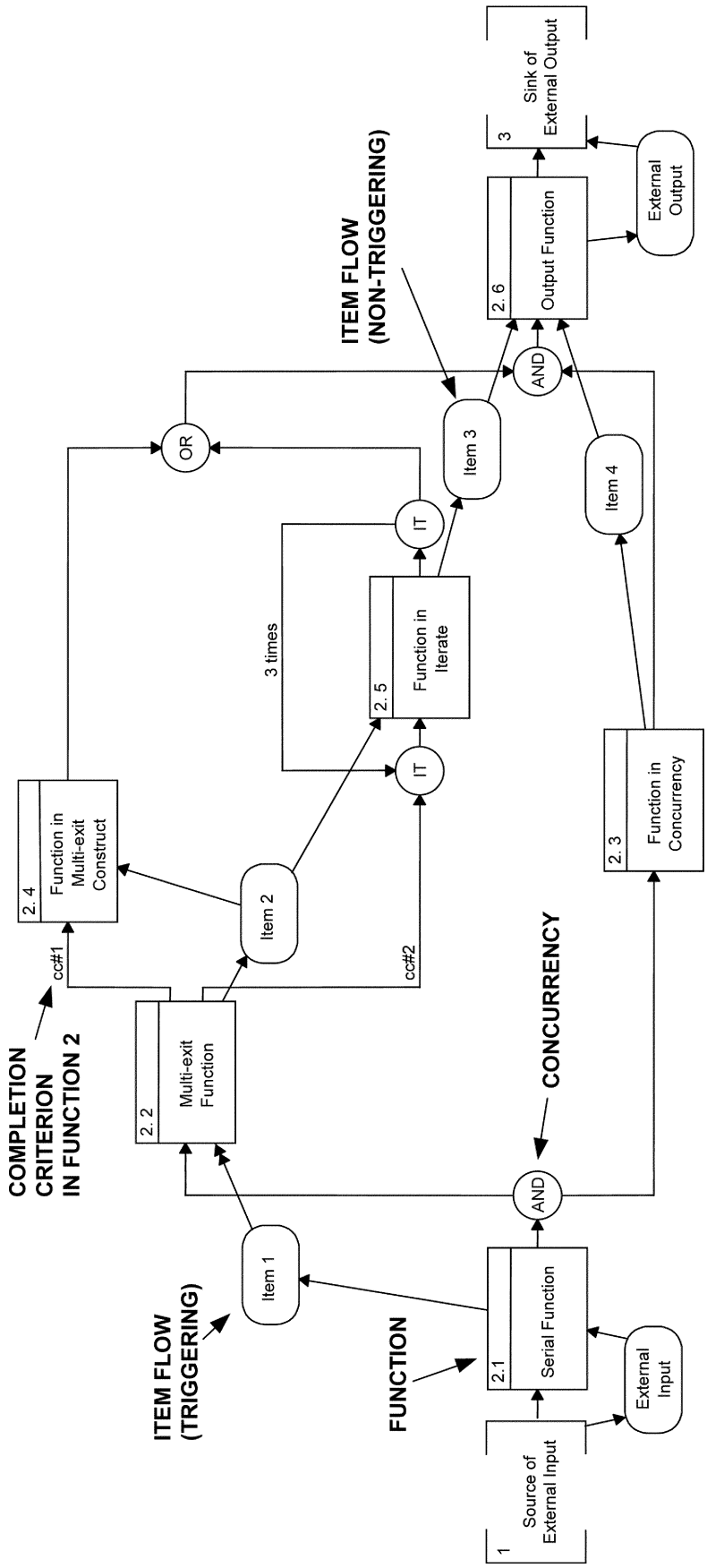


Figure 8. Enhanced functional flow block diagram for function 2 in Figure 7.

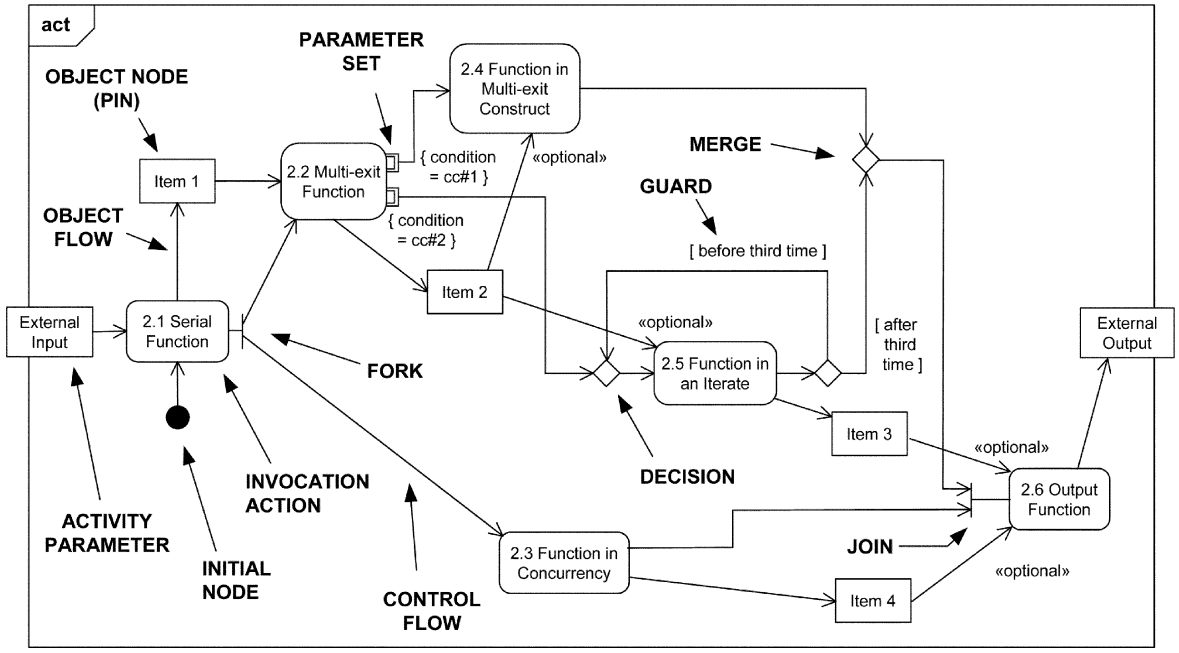


Figure 9. UML 2 Activity diagram corresponding to Figure 8.

sion and merge are matched one-to-one, as are forks and joins, accounting for the output parameter sets acting as decisions, and input parameters and control acting as a join.

3. All actions (activity usages) require exactly one control edge coming into them, and exactly one control edge going out, except for activities with parameter sets (multi-exit).
4. All control is enabling (see control values in Section 5.1.1).
5. All control flows into an action support control buffering (see control pins in Section 5.1.2).
6. Buffering of items and control is a queue (first-in, first-out).
7. Object flow is never used for control (see control parameters in Section 5.1.1), except for pins of parameters in parameter sets.
8. A maximum of one item instance is input or output per parameter per activity execution. In UML, this means the maximum multiplicity of parameters is 1.
9. All outputs are required. In UML, this means output parameters have a minimum multiplicity of 1. In SysML this means the «optional» stereotype cannot be applied to output parameters.
10. Parameters cannot be streaming (see Section 5.1.1).[16]
11. Parameters sets only apply to output parameters (multi-exit). Parameters are not shared across parameter sets. Parameter sets have exactly one parameter. If one parameter is in a parameter set, then all the output parameters on the activity are.
12 Edges (flow lines) cannot have time constraints.
13. The following SysML stereotypes cannot be applied: «rate», «controlOperator», «noBuffer», «overwrite» (see Sections 5.1.2 and 5.2.2).

SysML also provides a «probability» stereotype that can be applied to output parameter sets (multi-exit functions in EFFBD) and to edges going out of decision nodes or object (select nodes or item nodes in EFFBD). The modeler can specify the probability of an output or flow path being chosen during execution.

4.2. EFFBD Pattern Translations to SysML and UML 2 Activities

The remainder of the translation between EFFBD and SysML/UML 2 Activities requires multiple elements of UML 2, as summarized in Table III, and shown in the figures of this section.

EFFBD uses a loop node to start and end a flow cycle, whereas UML uses a merge node to start a cycle and a decision node to end it. Some implementations of EFFBD have a special node to exit a loop, which transfers control to the function after the closing loop

[16]UML also supports exception output parameters, which are not part of EFFBD.

Table II. EFFBD, UML 2 Activity, and UML SE RFP Requirements

	EFFBD	UML 2 Activity	UML SE RFP Requirement
Function		Activity Execution	Activation, 6.5.2.2, 6.5.2.3, 6.5.2.4.1
	Function	Activity	6.5.2.1.3 a-b,e,g-i,k
		Action (Activity Usage)	
	External Input/Output	Activity Parameter Node	Function Port: 6.5.2.1.1, 6.5.2.1.3 c,d
Item	Item Flow	Object Flow	6.5.2.1.3 f,g,j
	Item Node	Pin (Item Usage by Activity Usage)[1]	
	Triggering Item Input	Required Parameter	Special case of 6.5.2.2.2, 6.5.2.2.3
	Non-triggering Item Input	Optional Parameter[2,3]	Special case of 6.5.2.2.2, 6.5.2.2.3
Control	Control Flow	Control Flow	6.5.2.2.1
	Select	Decision, Merge	6.5.2.2.2 c
	Branch Annotation	Guard	Special case of 6.5.2.2.2, 6.5.2.2.3
	Concurrency	Fork, Join	6.5.2.2.2 c
	Multi-exit Function	ParameterSets[4]	Special case of 6.5.2.2.2, 6.5.2.2.3
	Completion Criteria	Postconditions on parameter sets[5]	Special case of 6.5.2.2.2, 6.5.2.2.3
	Iteration, Loop	Flow, Decision, Merge	6.5.2.2.2 c

[1]An earlier article [Bock, 2003a] incorrectly translated an item node with multiple outgoing flows as a UML object node followed by a UML fork. Multiple flows from an item node translate directly to multiple flows from a UML object node, as shown in Figure 9 for ITEM 2. In EFFBD and UML this means the item flows to either 2.4 or 2.5, but not both. See discussion of item instance in Section 3.3.

[2]An earlier article [Bock, 2003a] incorrectly translated nontriggering inputs to UML streaming inputs. EFFBD does not include streaming inputs. Nontriggering inputs are those that are not required to be present for the function to be enabled.

[3]An optional parameter in UML has a lower multiplicity of 0. SysML provides a stereotype «optional» on parameters to apply this constraint.

[4]Before finalization, UML 2 only supported parameters sets for items, not control. With the finalized UML 2 and SysML, parameters can carry control and, like all parameters, be grouped in parameter sets.

[5]Before finalization, UML 2 only supported postconditions on parameters. The finalized UML 2 supports them on parameter sets also.

Table III. Translation of EFFBD and UML Activity Patterns

EFFBD	Activity Diagram
Iteration, Loop	Pattern of Flow, Decision, Merge
Kill branch	Pattern of Fork, Join, Interruptible Region, and Join Specification
Iteration/Loop Queuing	Pattern of Central Buffer Nodes

node, whereas UML uses a decision node and outgoing flow to do this. These EFFBD constructs are translated to the pattern at the bottom of Figure 10. A decision can exit the loop at any point, and there can be multiple exit points. Loop nodes determine whether to exit the loop at each cycle.

EFFBD iteration nodes calculate the number of times to cycle only once, at the beginning of the first iteration. This aspect of EFFBD semantics is translated

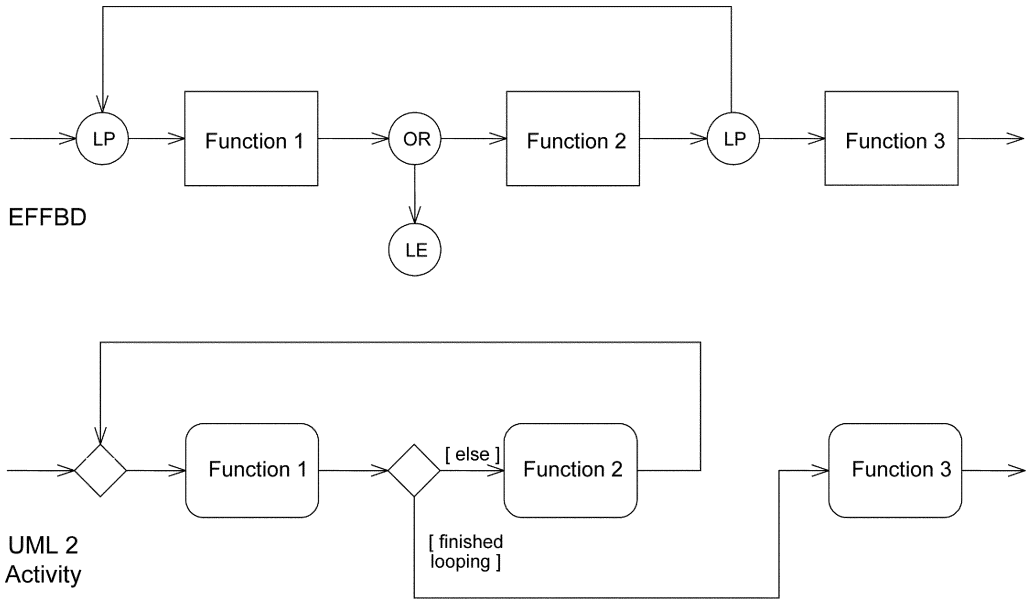


Figure 10. Loop.

to the pattern at the bottom of Figure 11, using an additional activity and a combined decision/merge node at the beginning. The activity determines how many times to cycle once at the beginning, and the decision node tests against this at each cycle.

EFFBD kill branches are flow lines going out of start-concurrency nodes indicating that if the branch reaches the corresponding end-concurrency node before the other branches from the same start-concurrency, then the others are terminated. This aspect of EFFBD semantics is translated to the pattern shown at the bottom of Figure 12. The dashed box indicates an interruptible region, and the edge labeled A is the interrupting edge. If edge A is traversed before FUNCTION 1 is complete, then FUNCTION 1 will be terminated. Otherwise, both flows from the fork complete, and are

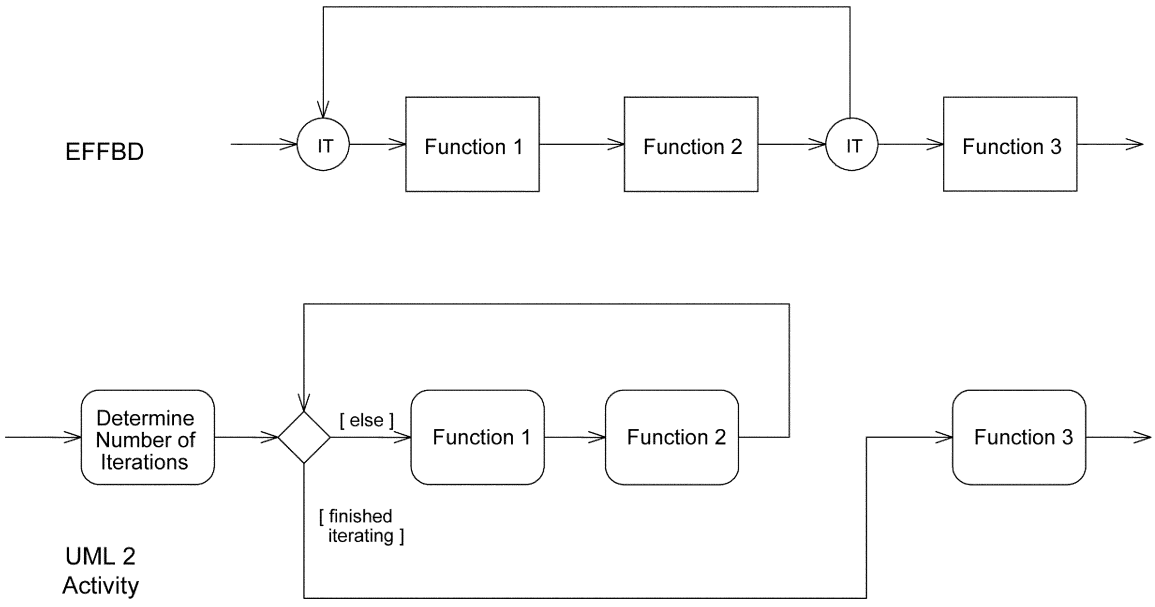


Figure 11. Iteration.

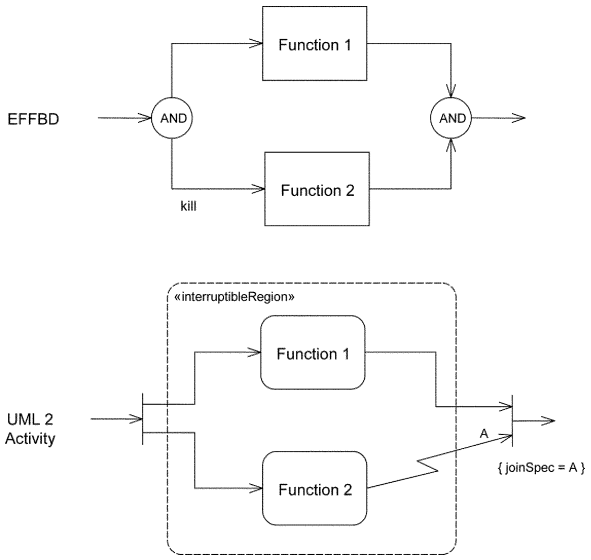


Figure 12. Kill branch.

synchronized at the join. The join specification requires only that edge A be traversed, so will be satisfied whether edge A terminates FUNCTION 1 or not.[17]

EFFBD iteration and loop nodes support queuing, whereas UML decision nodes do not. This aspect of EFFBD semantics translates to a UML central buffer node before the iteration or loop node, as shown in Figure 13. A central buffer node is a buffer that is not attached to an activity usage. It holds control and items coming into it, until they can flow downstream.

4.3. Execution Semantics for EFFBD Activity Execution in UML

EFFBD is executable and supports tools that produce timing information. It is important that EFFBD and the translations to UML 2 Activities in Sections 4.1 and 4.2 result in exactly the same execution traces. The points below are the execution rules for UML 2 Activities that apply to SysML's interpretation of EFFBD (excluding replication and resources, see introduction to Section 4).[18]

- Before activity execution: An activity usage (UML action) waits for all required item inputs (EFFBD triggering inputs), in whatever quantity is required, and waits for all control inputs required, then begins. Optional inputs (EFFBD nontriggering inputs) are accepted if they arrive before the activity starts.
- During activity execution: Items arriving at inputs while the action is executing are queued. EFFBD does not include streaming, which means no inputs are accepted by an activity usage if the activity is already executing (see Section 5.1.1).[19] Incoming enabling control is queued in a first-in, first-out buffer. EFFBD does not include disabling control, which would abort the execution when it arrives (see Section 5.1.1).
- After activity execution: An activity usage provides all required item outputs, and all control outputs for outgoing control flow lines. If output parameters are grouped into parameter sets (EFFBD multi-exit), this rule applies to exactly one of the parameter sets, as determined by the activity execution. No outputs are provided on parameters in other parameter sets.

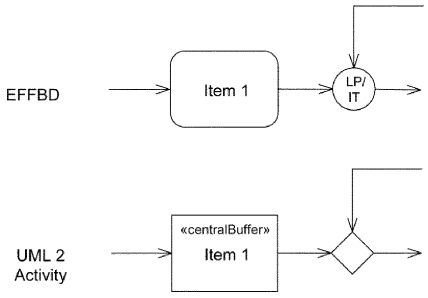


Figure 13. Iteration/Loop queuing.

More information on UML 2 Activity execution semantics is available in a separate series of articles [Bock, 2003c, 2003d, 2003e, 2004a, 2004c].

5. STREAMING ACTIVITIES

Streaming activities are at the end of the spectrum in Section 2 where activities accept input and provide output while they are executing. This compares to non-streaming activities of Section 4, which accept input only when they start, and provide output only when they complete, as in EFFBD. Since streaming activities can potentially operate indefinitely, working on inputs they receive over time, it is useful to have additional control capabilities to determine when they should terminate. This is described in Section 5.1. And since streaming activities can accept item inputs while they execute, it is not necessary to buffer inputs while waiting for the

[17] If there were a second kill branch, with an edge named B leading to the join, then the join specification would be "A or B."

[18] EFFBD requires that resources are available for execution to begin and the resources are returned upon completion of execution.

[19] UML supports concurrent execution of queued inputs (reentrancy), which is not included in EFFBD.

activity to complete. This makes them particularly suited to systems that have high or continuous rates of flow, covered in Section 5.2.

5.1. Additional Control Capabilities

This section describes control capabilities needed for activities to terminate other activities. These capabilities are useful for disabling streaming activities that would otherwise run indefinitely. For example, the engine in a car operates until the driver turns it off. This section also covers the related feature of control buffering, because it is easier to understand with the others. In summary, the features are:

- Multiple control values (6.5.2.2.1 b–d): These values flow along control flow lines in an activity diagram. The values include disabling, to stop an activity that is already executing, as well as the enabling control value to start an activity. For example, a disabling control value can be sent to a heating activity when the temperature goes above a certain threshold, to discontinue heating. An enabling control can be sent when the temperature goes below the threshold, to restart heating.
- Control operators (6.5.2.2.2): These are a kind of activity that can accept control values as input or provide control values as output, to enable or disable other activities. For example, a control operator may monitor temperature and output enabling and disabling control for a heating activity. Control operators are not enabled or disabled by the control values on which they operate. They can support complex logic for determining when other activities start and stop.
- Buffering control (6.5.2.1.3 d): It is sometimes useful to keep control values until an activity is already enabled or disabled, for use when activity is ready to accept them. For example, the commands for taking radar readings may be queued if some of the readings take longer than expected.

Support for these features in EFFBD, UML 2, and SysML is shown in Table IV.[20] The first two are discussed in Section 5.1.1, and the third in Section 5.1.2.

[20]SysML and UML SE RFP requirements continue a long-term trend of unifying item flow and control flow. Modern flow models give item flow the power of control, by enabling activities when items become available (items are "pushed" to the activities from earlier function activations), as compared to traditional data store approaches that passively provide items on request from already enabled activities (items are "pulled" from the activities from passive data sources) [Bock, 2004a]. UML SE RFP requirements extend this by giving control the capabilities of item flow: multiple values, operating on values, and buffering values.

Table IV. Support for Additional Control Capabilities

	EFFBD	UML 2 Activity	SysML
Multiple Control Values	No	No	Yes
Control Operators	No	Partial	Yes
Control Buffers	Yes	Yes	Yes

5.1.1. Control Values, Control Parameters, and Control Operators

SysML provides for control values that enable or disable activities, and control operators that accept and emit these values based on potentially complex internal logic. Control operators can treat control as if it were an item, accepted it as input, or provided it as output through parameters, rather than being enabled or disabled it. Control values and operators are not included in EFFBD. UML 2 only supports one control value, for enabling an activity, but the finalized version allows extensions to other kinds of control.[21]

SysML supports control values and operators with a model library containing a classifier called CONTROLVALUE. The classifier has instances ENABLE and DISABLE, which are the individual *control values*. This classifier can be the type for parameters (item usages by activity definitions, see Section 3.3), making them *control parameters*. The values ENABLE and DISABLE can flow through an activity execution like any other item in the system. For example, Figure 14 shows control flowing from a temperature monitoring activity to a heating activity. The temperature monitoring activity emits a value of type CONTROLVALUE that controls the enabling and disabling of an activity for heating air. This makes it a *control operator*, which is highlighted by the stereotype.

The definition of the control operator MONITOR TEMPERATURE is an activity, as shown in Figure 15. It has an output parameter of type CONTROLVALUE, which is what makes the activity a control operator. The parameter is streaming, to provide outputs while the activity is executing, rather than just when the activity is done. The definition has a loop that measures the

Figure 14. Control value flow.

[21]UML 2 object nodes, including pins, can indicate that their type is to be interpreted as control, with the ISCONTROLTYPE metaattribute, so pins can carry instances of user-defined control types.

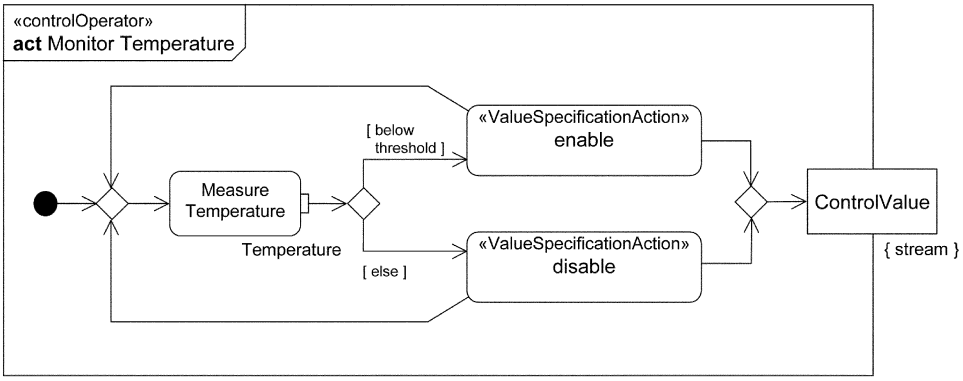


Figure 15. Definition of a control operator.

temperature, then outputs an ENABLE or DISABLE control value, depending on whether the temperature is below a threshold or not. The usage of MONITOR TEMPERATURE in Figure 14 has a pin (item usage on activity usage) corresponding to the output parameter. The pin is annotated with characteristics of the parameter, for readability. When a control value leaves the pin, it flows to the heating activity. A control value of ENABLE starts the heating activity if it is not already executing. If it is already executing, the enabling control value is ignored and discarded (see Section 5.1.2 for control buffering). A DISABLE control value disables the heating activity if it is executing at the time. If it is not executing, disabling control is ignored and discarded (see Section 5.1.2 for control buffering).[22] Future extensions could provide other control values, such as for suspension and resumption.

Control value parameters allow control operators to combine and manipulate control values without being enabled or disabled themselves. For example, in Figures 14 and 15 the control values enable and disable heating, not temperature monitoring, even though they come from temperature monitoring. This is because control values move through MONITOR TEMPERATURE via parameters and item flow, so are treated like any other item, whereas HEAT AIR has no parameter for control. Control values arriving at HEAT AIR have a direct effect on its execution, rather than simply moving through it as input and output.[23] An activity with a control value parameter could use the value to disable itself, but this is not required. Omitting the parameter guarantees the control value will start and stop the activity it arrives at.

A number of UML SE RFP requirements call for control operators of various kinds, as described below. SysML refined these requirements and satisfied them without using control operators.

- EFFBD selection and concurrency are supported by dedicated UML modeling elements for those purposes: decision and merge for selection, fork and join for concurrency, as shown in the Figure 9, rather than as usage patterns of control operators (requirement 6.5.2.2.2 c). Loops are supported in UML as patterns of decision and merges, also shown in Section 4.2. Users can define control operators for these, if desired, but the resulting model is more compact using dedicated elements, so no predefined control operators are provided in SysML.
- Multiple incoming control flows to an activity usage can be combined (6.5.2.2.1 a) using the UML join element with join specification (see example in Fig. 12), or can be preceded by a separate control operator that performs the combination. The control operator can also be embedded in the activity, but then it is hidden from diagrams that use the activity.
- Disabling an activity by timeout (6.5.2.2.1 e) is supported by a usage pattern of UML interruptible regions, accept event actions, and joins. An example of interruption by signal receipt is shown in Figure 33. Time event receipt is similar, but uses an hourglass notation.

5.1.2. Control Buffering

Control buffering is applicable across the spectrum of Section 2 when activities are disabled waiting on other inputs. Control arriving at an activity that is already

[22] Temperature monitoring activities would usually emit ENABLE and DISABLE values alternately, so the discard cases above would not occur. However, it is not possible to determine that the discard functionality is unnecessary from the above model. This would need to be declared by a process constraint on the function, or derived from more detailed modeling of MEASURE TEMPERATURE.

[23] Same for control values arriving at control pins (see Section 5.1.2).

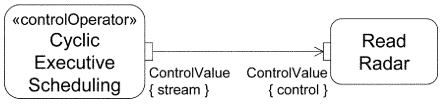


Figure 16. Control pin.

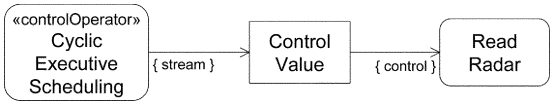


Figure 17. Control pin, alternative notation for Figure 16.

executing can be kept in a buffer until it restarts the activity. This is useful in situations where a task is started periodically (cyclic executive scheduling), but in some cases may take longer than the period to complete. Control buffering ensures that the activity is restarted when it is finished and catches up with the required period. For example, a radar device may receive control at regular intervals to take a reading, but some readings may take longer than normal. Control is queued by the device to catch up as much as possible with the desired timing. SysML supports control buffering, as does EFFBD and the finalized UML 2.

Buffering in UML 2 and SysML is specified at pins (item usages by activity usages, see Section 3.3), rather than parameters. The pin specifies whether and how control and items are buffered, rather than the parameter or activity. This gives flexibility in applying activities. Some usages can buffer and others not, while activities can be defined purely as transformations, without the burden of buffering issues. An activity can have its own internal buffers, but these are not visible from other activities that use it.[24] This is a refinement of UML SE RFP requirements (6.5.2.1.3 d).

Unlike regular pins, however, UML control pins do not pass their values into the activity; they only determine when the activity is enabled and disabled. Otherwise, the control value would be like any other item flowing through the activity via parameters, as in control operators. This means the activity definition has no parameter corresponding to the control pin. The example in Figure 16 shows a cyclic radar executive sending control to an activity through a control pin. If the reading activity is not finished when the executive sends another enabling control value, the value is buffered at the control pin until reading is finished, then starts the activity again.[25, 26] Compare this to Figure 14, which also has a control operator and a controlled activity, but would ignore and discard a redundant control value arriving at the controlled activity. Figure 17 shows an alternative notation that means the same thing as Figure 16.

Control pins and control parameters, described in Section 5.1.1, differ in that control pins accept control values that enable or disable their activity, and do not have a corresponding parameter, while control parameters have a corresponding pin at each activity usage, and support control values flowing into and out of an activity without necessarily enabling or disabling it. An activity may use a disabling control value parameter to disable itself, or it might not, but control values arriving at a control pin will always affect activity execution.[27] Control value parameters also do not specify buffering, whereas control pins do. Table V summarizes these characteristics.

Control flows without control pins have a limited buffering functionality in UML 2, because enabling control arriving at a disabled activity is held until the rest of the required inputs are available. However, if multiple enabling controls arrive at a disabled activity along the same control flow line, only the first one is kept. Control pins can be used to turn off even this limited buffering, by applying the SysML «noBuffer» stereotype to control pins (see Section 5.2.2). For example, Figure 18 shows an example of a sheet metal stamping activity. Enabling the activity requires control to arrive, which might be allocated to a button pressed by an operator. It also requires a piece of sheet metal on which to operate. However, if the sheet metal has not arrived, the control value should not be buffered, because stamping will start when the sheet metal arrives, and conditions may have changed in a dangerous way since control was provided. For example, the operator may have moved their hands from the button to clean the die. To prevent buffering, the «noBuffer» stereotype is applied to the control pin. Then control values are discarded if they arrive before the sheet metal does. A

Table V. Comparison of Control Pins and Control Parameters

	Control Pins	Control Parameters
Enable and disable activities	Yes	Depends on activity definition
Specify control buffering	Yes	No

[24] Activity definitions have parameter nodes, one for each parameter, that provide internal buffering [Bock, 2004a].

[25] In EFFBD, buffering is a queue (first-in, first-out), and UML 2 defaults to the same.

[26] A disabling value sent to a control pin might prevent the target function from ever starting again, because the usage needs to receive an enabling control on all incoming flow lines to start. The only way an enabling control can get past the disabling one in the buffer is if there is a sorting function on the buffer that chooses the later enabling control value over the earlier disabling control value.

[27] Assuming they are not discarded (see Section 5.2.2).

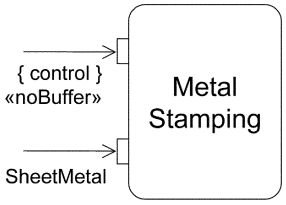


Figure 18. Control pin with «noBuffer».

control value arriving after the sheet metal immediately enables the stamping activity.[28]

5.2. Continuous Systems

This section describes concepts and capabilities that are useful across the spectrum of Section 2, but especially for streaming activities and what might be loosely called "continuous" systems. For example, specifying the allowable rate of flow for a liquid from a pumping activity is important in properly applying that activity. One might call this a continuous system, but does "continuous" mean the pumping activity is always enabled, or the flow rate varies continuously over time, or both, or something else? Section 5.2.1 identifies the various kinds of continuity, and Section 5.2.2 describes how SysML and UML 2 support them.

5.2.1. Defining Continuity

The term *continuous* is used in a variety of ways, each of which must be defined separately to determine how it should be applied. Five meanings were identified during the refinement of UML SE RFP requirements on continuity (6.5.2.1.3 j), as illustrated in Figure 19, and described below.

1. Mathematical properties:
a. Continuous type: a characteristic of ordered sets that every two elements have at least one other element between them, as illustrated in Figure 20. For example, the real numbers are a continuous type, because any two real numbers have another real number between them.[29]

[28]This might be specified with a state machine where some states ignore the start button, but this does not scale well to multiple inputs. The more inputs, the more states are required for the combinations of arrived inputs, and transitions on each state for each input.

[29]The mathematical term is *dense*. Continuous and dense are actually distinct mathematically. For example, the rational numbers are dense, because every two rational number has another one between them, but they are not continuous, because there are nonrational numbers between any two rational ones (irrational numbers). This article takes density as the criteria for continuity 1a, because it is enough to support system specification to as fine a granularity as necessary.

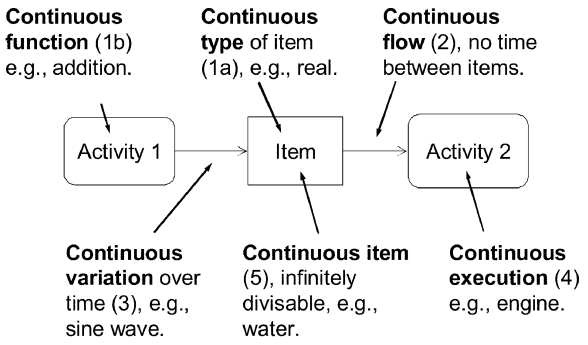


Figure 19. Kinds of continuity.

b. Continuous function: a characteristic of activities with numeric inputs and outputs in which smaller changes in the inputs correspond to smaller changes in outputs, as illustrated in Figure 21. For example, the equation $y = 2x$ gives y as a continuous function of x, because the smaller the changes are in x , the smaller the changes are in y. Continuous functions necessarily require values of continuous types as input and output, because the changes in input can be arbitrarily small.
2. Continuous flow of items or control: a characteristic of items or control that pass along a flow line with no discernable time between them, as illustrated in Figure 22. For example, oil flowing to a pump is a continuous flow, because the time between the fluid elements is indiscernibly small.
3. Continuously varying inputs or outputs: a characteristic of inputs or outputs of values that vary as mathematically continuous functions (1b) of time, as illustrated in Figure 23. The smaller the change in time, the smaller the change in value. For example, a voltage at an input may vary as the sine of time.
4. Continuous execution: a characteristic of activities that run as long as their system or subsystem is running. For example, an engine in a car, as illustrated in Figure 24.
5. Continuous item: a characteristic of an item that it can be divided as much as needed by the application and result in a continuous item of the

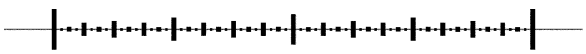


Figure 20. Continuous type.

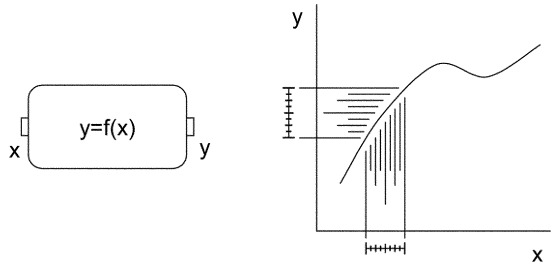


Figure 21. Continuous function.

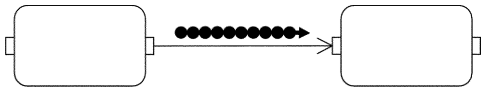


Figure 22. Continuous flow.

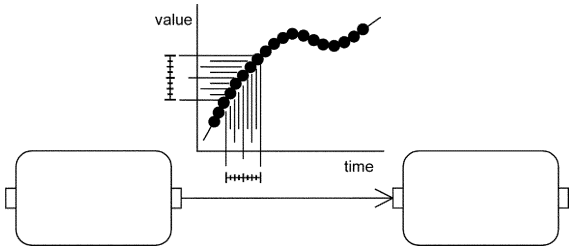


Figure 23. Continuous variation.

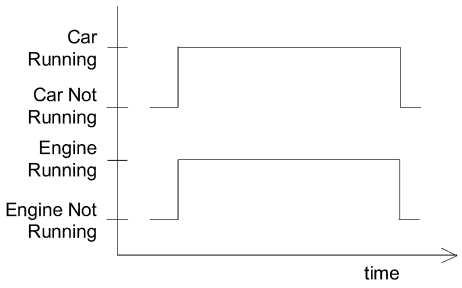


Figure 24. Continuous execution.

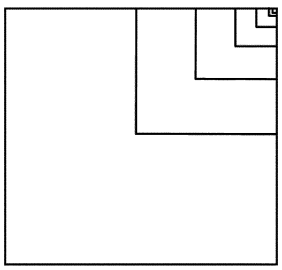


Figure 25. Continuous item.

same kind, and where the identity of its constituents is not important for the application: for example, a tank of water or bucket of ball bearings.[30]

The kinds of continuity identified above are mostly independent of each other, as shown by the blank cells in Table VI. For example:

- Values of a continuous type (1a) may flow noncontinuously (2). For example, an activity may output a temperature measurement as a real number, but is only do so every 10 minutes.
- A flow may be continuous (2) for values of noncontinuous types (1a). For example, values flowing between one activity and another may be only be true or false, and still flow with no discernable time between them.
- An activity may be a continuous function (1b) but receive or produce noncontinuous flows (2). For example, an activity may produce the sine of its input, but the input flow might arrive only every ten seconds.
- An activity may be a continuous function (1b) but receive or produce noncontinuously varying values (3). For example, an activity may produce the sine of its input, but if the input values are integers, the output will be the sine of integers, which will not vary continuously.
- Input or output values may vary continuously with time (3) for activities that are not continuous functions (1b). For example, the input to an activity may be a sine wave voltage, but it may output a voltage of positive or negative five, depending on whether the input is above or below a certain threshold value. A small change in input can cause a large change in output.
- Inputs and outputs can be of continuous types (1a) for activities that do not execute continuously (2). For example, an activity that multiplies a real number by two might be executed only once during the life of a running system.
- An activity may have continuous input and output flows (2) and not be executing continuously (4). For example, an activity that multiplies a real number by two may be invoked repeatedly during the system lifetime on inputs arriving continuously from another source. The flows in and out will be continuous, causing concurrent executions of the activity, but the activity will be start-

[30]Continuous items do not assume an ordering among individual elements, as continuous types do.

Table VI. Dependencies between Kinds of Continuity

	Type (1a)	Function (1b)	Flow (2)	Variation (3)	Execution (4)	Item (5)
Type (1a)		←		←		←
Function (1b)						←
Flow (2)				←		
Variation (3)						←
Execution (4)						
Item (5)						

ing and stopping repeatedly, rather than always executing.

- An activity may execute continuously (4) and have noncontinuous inputs and output flows (2). For example, the temperature monitoring activity in Figure 15 of Section 5.1.1 might operate all the time, but only outputs control values to the heating activity at a noncontinuous rate, depending on how fast the temperature is measured and compared to a threshold.
- An activity may operate on continuous items (5) and have noncontinuous input and output flows (2). For example, a water heater might take buckets of water as input, rather than having it piped in continuously. Continuous items can be packaged in separately identifiable batches.

The arrows in Table VI identify how the kinds of continuity depend on each other, with the arrow pointing from dependent kinds to others they require. Only six of the thirty combinations are dependent, almost all of which involve continuously varying inputs and outputs (3) or continuous items (5). The six constraints are:

- A continuous function (1b) requires continuous types (1a) for input and output, since the changes in the inputs and outputs can be arbitrarily small.
- Continuously varying inputs and outputs (3) require continuous types (1a) to describe values changing continuously with time.
- Continuously varying inputs and outputs (3) require continuously flowing values (2) to describe values changing continuously with time.
- Continuous items (5) require continuous types (1a) to describe amounts of material.
- Continuous items (5) usually require continuous activities (1b), since arbitrarily small changes in input will usually cause arbitrarily small changes in output.
- Continuous items (5) usually require continuously varying inputs and outputs, (3) since the items are described with amounts of material, and this varies continuously with time.

The kinds of continuity described above are also independent of UML's streaming parameters, as in Figure 14, despite the name. Streaming parameters declare that an activity might provide outputs or accept inputs while it is executing, rather than just at the start and finish of execution. Streaming parameters do not require that an activity accept continuous flows as input or provide continuous flows as output during execution. For example, the temperature monitoring activity in Figure 15 has a streaming output, but outputs control values at a rate that depends on how long it takes to measure the temperature. Streaming parameters also do not require any particular execution of the activity to actually accept input or produce output while executing at all. For example, a temperature monitoring activity might be defined to only output a control value when the temperature crosses the threshold, which never happens if the threshold is low and the activity is operating in a warm climate.

5.2.2. SysML and UML 2 Support for Continuous Systems

SysML and finalized UML 2 models support most of the meanings of continuity in the last section. The first mathematical property in the previous section (1a) is partially supported in SysML with a primitive datatype for real numbers. The second (1b) is covered by func-

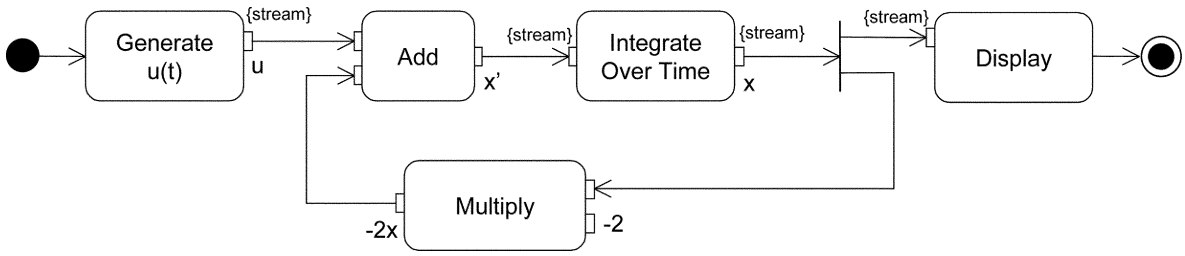


Figure 26. Activity for numerical solution of Eq. (1).

tion behaviors in the finalized UML 2, which are behaviors that operate only on data types, where the output values depend only on the input values, and have no other effect than to compute output values. Function behaviors can be specialized if it is required to classify some of them as mathematically continuous.

UML 2 and SysML support continuous flows (2), continuously varying inputs and outputs (3), and continuous execution (4). There is no limit on how close together in time items may move through an activity, how different one input or output value is from the next one in time, or how long activities can execute.[31] SysML treats continuous flow (2) as a flow where the time between individual items or control values is effectively zero as far as the application is concerned. Continuously varying outputs (4) are specified with activities that continuously generate values according to mathematically continuous functions. UML 2 and SysML also do not completely prescribe the timing of flow through an activity. In particular, flows on separate lines may be coordinated to occur so that each subactivity is executed once in a cycle, if that is required by the application.

Figure 26 illustrates the flexibility above, and gives an example of an application from the middle of the spectrum in Section 2. It shows an activity diagram of the ordinary differential equation in Eq. (1), adapted from [MathWorks, 2004], omitting the item types on flows for brevity.

$$x'(t) = -2x(t) + u(t) \tag{1}$$

The activity on the left generates values for the mathematical function u. This is a streaming activity, as indicated by the streaming output of values produced while it is executing. It is started once and feeds values to the addition function. The addition activity is nonstreaming, since it is enabled only when it receives a value on each input, and does not accept any more input until it is finished and provides an output (EFFBD functions also require control, see Section 4.1, which is omitted here for brevity). The first execution of ADD will require a value on the input from MULTIPLY, even though that has not executed yet. This is provided by a default value of zero for the input specified in ADD. Then the sum is passed to an integrator, which is a streaming activity.[32] It needs to execute for as long as it accepts inputs, because it retains the inputs and outputs of the previous cycle to calculate the result for the current one. The integrated value is passed to a display activity, also streaming, because it maintains a graph of the result over time. The value is also passed to a multiplication function. Like addition, this is a nonstreaming function.

Figure 26 could execute each activity once in a cycle, as in some implementations of time march algorithms, such as Runge-Kutta [Lambert, 1991]. Each clock cycle represents the increment of time chosen for the solver. This has the advantage of preventing backup of values at the slower activities; for example, integration may be slower than multiplication. If some activities produce a value earlier than others in any particular clock cycle, their output does not flow to the next until the slowest activity is done. This technique also allows the streaming activities to access a global value for time that will be synchronized with whatever values are given as input.

The example above illustrates that backup is a general issue for applications that use nonstreaming activities, or streaming activities that are not always executing. Continuous systems are especially prone to backup because there is no limit on how fast inputs arrive at an activity usage. A nonstreaming activity may still be operating on a previous input, or a streaming activity may be disabled. This will cause the inputs to be buffered, and if the rate of arrival stays high, will

[31] An earlier article [Bock, 2003a] raised an issue with coordinating continuously flowing inputs at an activity usage. This is not a problem when continuous flow is treated as a limiting case of discrete flow.

[32] The current time and the time increment could also be given as input to the streaming activities when they start. This would make them hybrid EFFBD/interaction activities, because some parameters would be streaming and others would not.

cause a backup that cannot be cleared until values stop arriving.

SysML and UML 2 provide a number of features to address backup, summarized in these bullets and described below:

- SysML provides for specifying rates at which items and control arrive and leave activities, including continuous rates («rate» stereotype). This helps determine where a model might have bottlenecks,
- Values can be discarded («noBuffer» and «overWrite» stereotypes), to reduce buffer size.
- A usage of an activity can have multiple executions of that activity (reentrant activities), to clear backups.

The «rate» stereotype specifies the rate at which items or control move through an activity. It may be applied to:

- Streaming parameters to specify the possible rate of flow of items or control in or out of streaming activities, independently of how the activity is used.
- Flow lines in an activity to specify the possible rate of flow of items or control between activity usages in a diagram.

The «rate» stereotype has a RATE property that can specify a particular flow rate, or a range or other constraint on acceptable flow rates.[33] The rate stereotype is specialized to a «continuous» stereotype for continuous flows and «discrete» for noncontinuous flows. For example, Figure 27 shows a version of Figure 4 where the «continuous» stereotype appears on an object node. Object nodes represent the types of items flowing through parameters (see Fig. 5), so the stereotype applies to the parameters of HEAT LIQUID and DISPENSE WATER. All the «rate» stereotypes, including «continuous», only apply to parameters if the parameter is streaming. The notation in Figure 27 is a way to show this detailed model in a more concise way than with annotated pins, as in Figure 28.

Figure 28 is a version of Figure 14 where «rate» is applied to the streaming output parameter of MONITOR TEMPERATURE, restricting the outflow rate to below one control value every 5 minutes (UML supports showing the properties of parameters on the corresponding pins, as well as displaying the stereotype property without the showing stereotype, when it is unambiguous). This parameter rate constrains flow for all activities that use

[33]SysML and UML 2 provide ways to express values for flow rates, such as 4 liters per minute, and constraints on flow rates (see Footnote 13).

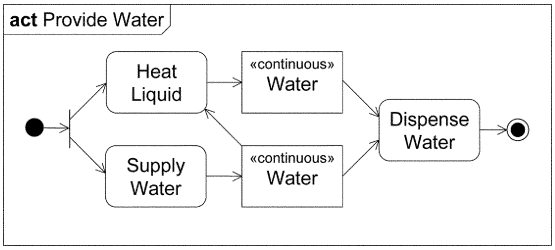


Figure 27. Continuous rate appearing on object node.

MONITOR TEMPERATURE. In its particular usage in Figure 28, the modeler happens to know the environment temperature is stable enough that the actual outflow will be less than one every 10 minutes. This only applies to this particular usage of MONITOR TEMPERATURE, not in general. The flow rate may be different in other activities that use MONITOR TEMPERATURE. The global rates on parameters of MONITOR TEMPERATURE must be compatible with the local rates on flows of its usages, which they are in this example.

Consistency rules apply to rates specified in an activity. For example, rates on control flow going out of an activity usage must be the same or less than the lowest of the rates on control flow lines coming into the usage. The same applies to rates on item flow going out of nonstreaming, required output parameters on an activity usage, which must be the same or less than the lowest of the rates on incoming flow lines for nonstreaming, required input parameters. They can be less only if the activity can be disabled, or if input items can be discarded by limitations on buffering. Two other rules can be derived from this:

- An activity usage with only continuous flows into nonstreaming input parameters can only have continuous flows out of its nonstreaming output parameters.
- An activity usage with noncontinuous and continuous flows into nonstreaming, required input parameters can only have noncontinuous flows out of its nonstreaming, required output parameters.

Another way to address backup problems with SysML is to reduce buffering by discarding items (see Section 5.1.2 for the basics of buffering). This is natu-

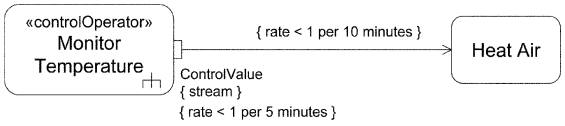


Figure 28. Flow rate on streaming parameters and flow lines.

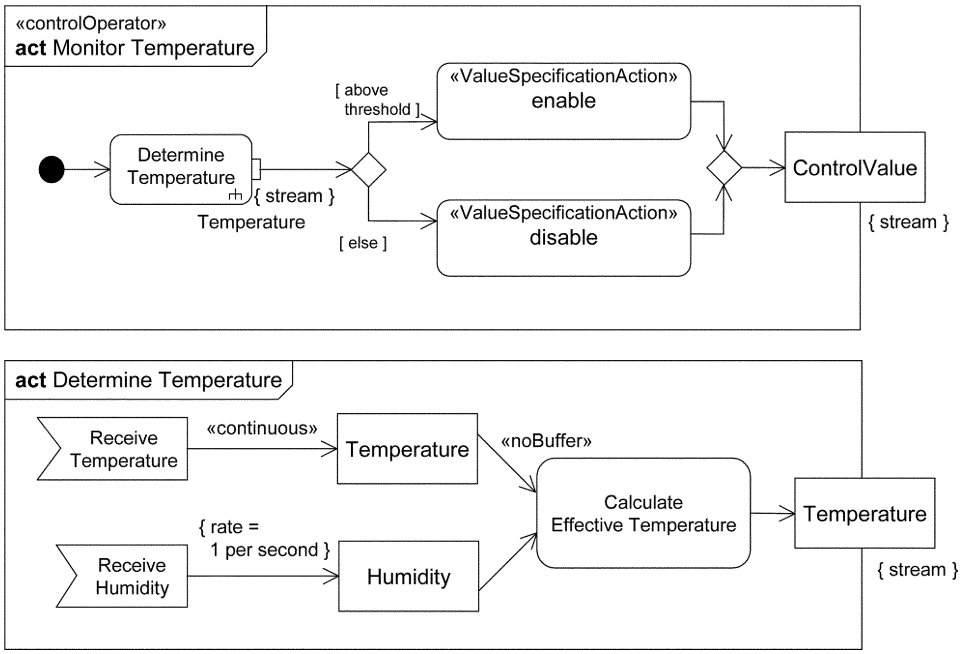


Figure 29. Using «noBuffer» with continuous and discrete flows.

rally restricted to applications where items can be destroyed, such as information or ephemeral physical characteristics such as voltage levels. SysML has two stereotypes for discarding items from buffers:

- «noBuffer»: discards values arriving at an input pin if the activity usage cannot accept them immediately.
- «overWrite»: discard values in a full buffer to hold newly arriving values. The buffer can be any pin or central buffer.

Figure 29 shows an application of «noBuffer» to prevent backup of a continuous flow. It is an "open loop" version of the example in Figure 15 of Section 5.1.1. The DETERMINE TEMPERATURE activity generates a stream of effective temperatures based on the actual temperature and humidity arriving from external sources, indicated by the receive actions (flag shapes) at the lower left. Humidity information arrives at a noncontinuous rate, while temperature arrives at a continuous rate. To prevent temperature values from backing up at the input pin to CALCULATE EFFECTIVE TEMPERATURE, the «noBuffer» stereotype is applied. This pattern can also model electrical and other transient signals that are not buffered at the receiver, as well as ensure control values have an effect at exactly the time they arrive, rather than later, as in Figure 18 of Section 5.1.2.

Figure 30 shows an application of «overwrite» to ensure the latest arriving values are buffered in applications where data can go "stale." It is a variation of DETERMINE TEMPERATURE from Figure 29 with discrete flows of different rates into the effective temperature calculation. The input buffer for temperature is set

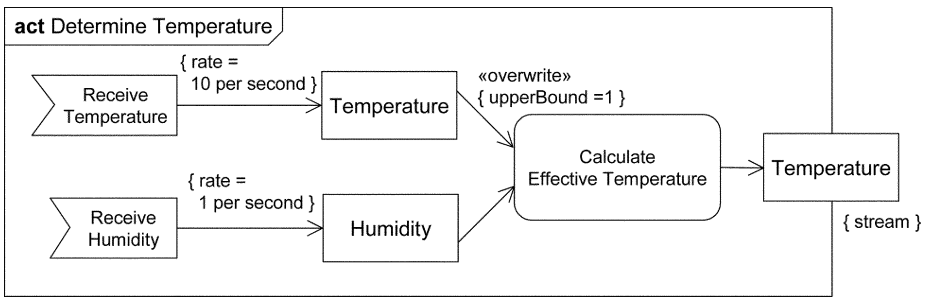


Figure 30. Using «overwrite» with mismatched discrete flows.

to hold a maximum of one value, with the UPPERBOUND property. If the buffer already has a value when a new one arrives, the «overwrite» stereotype indicates that the old value should be discarded to make room for the new one. When a humidity value arrives, it will be paired with the most recent temperature as input. The stereotypes «overwrite» and «noBuffer» have the same effect when applied to continuous flows, because a continuous flow will always provide the latest value regardless of which stereotype is applied. However, if the flow is sometimes continuous and sometimes discrete or not flowing at all, «overwrite» will keep the last value, while «noBuffer» will not.

Another technique for preventing backup is to use a feature of UML 2 that provides for concurrent executions of the same activity at the same usage (also see EFFBD replication, see Section 7). If an activity is *reentrant*, each complete set of inputs arriving at a usage will create a new concurrent execution of the activity. It is assumed these activities can be executed concurrently without any conflict between the multiple executions. For example, in Figure 26, addition and multiplication are reentrant, so a new concurrent execution will be started for newly arriving inputs, even if the operation on previous inputs is not completed yet. Outputs will be produced as fast as the inputs arrive, with only a single operation delay between the acceptance of an input and production of the corresponding output. The only limit is in how many concurrent executions the underlying execution engine can support.

Continuous items (5), the last meaning for continuous in Section 5.2.1, are partially supported in SysML and UML 2. They are modeled as classes (blocks in SysML), like any kind of item. For example, a class WATER can represent the general characteristics of water. It is expected that instances of these classes will be identifiable quantities of water, for example, the water in a particular tank, bottle, or even individual molecules. UML also does not restrict how small an item may be, for example, to model continuous material or energy flow. However, a collection of water molecules has different characteristics from individual ones, such as boiling point of the collection, and molecular weight of the individual molecule. And the water in a tank can be divided and still be water, whereas an individual molecule cannot. These can be distinguished by the units of measurement.

6. ACTIVITY DECOMPOSITION AND ALLOCATION

This section covers two common aspects for presenting activities: activity decomposition and allocation by partition. Activity decomposition corresponds to what is typically called functional decomposition, described in Section 6.1. UML partitions provide for dividing up the subactivities of an activity for various purposes, one of which is to allocate subactivities in the partition to structural elements, as described in Section 6.2.

6.1. Activity Decomposition

A commonly used SE diagram shows how activities decompose into others, omitting flows between subactivities. For example, the activity decomposition for Figures 14 and 15 in Section 5.1.1 would show the activity in the first diagram breaking down into MONITORING TEMPERATURE and HEAT AIR, and MONITORING TEMPERATURE breaking down into MEASURE TEMPERATURE. These diagrams show only decomposition, without other information normally shown on activity diagrams, such as flow lines between usages, and coordinating nodes such as decision branches.

Activity decomposition can be modeled with class diagrams (block definition diagrams in SysML), using UML 2's application of class and instance concepts (item definition and item instance) to activity and execution (activity definition and execution instance):

- In UML and all class-based languages, instances must conform to their class. For example, the individual water heater with serial number 234523 must conform to the class WATER HEATER, by having values for the attributes specified by the class, such as CAPACITY, that are within the ranges defined by the class, for example, a positive integer. An individual water heater with a negative value for capacity would not conform to its class.
- UML 2 recognizes the same idea applies to activities, in particular, activity executions must conform to their activity definition. For example, every execution of MONITOR TEMPERATURE in Figure 15 must conform to the control and data flow shown in that diagram. An execution that did not measure temperature and test it against a threshold would not conform to MONITOR TEMPERATURE. This means activities can be treated as classes, with their executions as instances.
- Classes can have associations between each other. Associations specify the allowable links that can exist between instances of the associated classes. For example, the class WATER HEATER might be associated with a class POWER SOURCE, which means individual water heaters will be linked to individual water heaters.
- UML provides for strong composition associations, which means destroying an instance at the

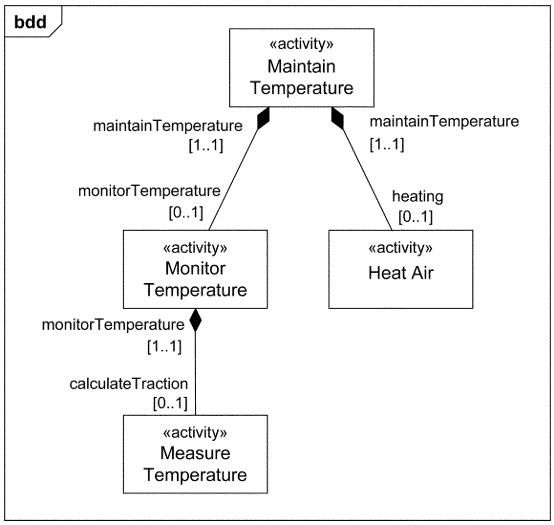


Figure 31. Activity decomposition.

whole end destroys instances at the part end [Bock, 2004b]. For example, a strong composition association between WATER HEATER and WATER TANK classes, with the heater on the whole end, means destroying an individual water heater causes the destruction of its water tank.

Applying strong composition associations to activity classes, the termination of execution of an activity on the whole end of a link will terminate executions of activities on the part end of the link.[34] For example, Figure 31 shows the activity decomposition for Figures 14 and 15. using a SysML block definition diagram (class diagram in UML). The «activity» keyword is applied to the classes to show when they represent activities. Executions of MAINTAIN TEMPERATURE will start executions of MONITOR TEMPERATURE and HEAT AIR. These subexecutions are linked to their parent by instances of composite associations, notated with black diamonds at the whole end of the composition. In functional terms, this means disabling an execution of an activity will disable executions of its subactivities. For example, disabling an execution of MAINTAIN TEMPERATURE will disable executions of MONITOR TEMPERATURE and HEAT AIR that happen to be executing under it at the time.

The numbers on each association specify how many of the instances of the class on the numbered end can link to one instance of the other end. For example, an execution of MAINTAIN TEMPERATURE can relate to no more than one instance of MONITOR TEMPERATURE. This constrains the number of concurrent synchronous executions of the activity that can be invoked by the containing activity (partially addressing replication, see Section 7).

[34]Composition does not specify the order in which executions on the part end will be terminated.

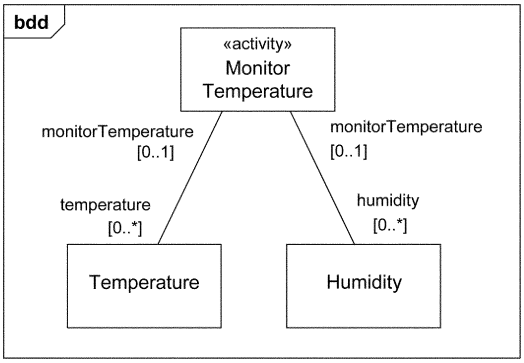


Figure 32. Items on a class diagram (block diagram in SysML).

SysML also extends activity notation to support activity usage names. These have the form "usage name : activity name." The usage name can be used on the ends of the associations in class diagrams to show which usage each subactivity corresponds to. One difference between classical functional decomposition and activity decomposition in class diagrams is that the same subactivity can appear more than once if it is used more than once in the containing activity.

Activities on class diagrams can also be used to show which kinds of items flow through them. For example, Figure 32 shows the kinds of items (classes) that flow through the open loop version of MONITOR TEMPERATURE in Figure 29. The associations are not composite, because disabling an execution of an activity does not necessarily destroy the items moving through it. For example, disabling a manufacturing process for cars does not necessarily destroy the parts moving through it.

6.2. Allocation with Partitions

Systems engineering addresses the relation of behavior to structure as part of an area of concern called *allocation*. SE requires flexible connection between behavior and structure because many alternative structures can support a required behavior or activity. This contrasts with common object-oriented software practice, in which a behavior is associated with a single class early in the development cycle [Bock, 2005a].

UML 2 provides a notation for showing structure on activities that is generalized in SysML for SE allocation. This is no longer a pure activity diagram, because it relates activity to structure, and is sometimes called a "swim lane" diagram. Figure 33 shows an example from the SysML specification with UML 2 partitions

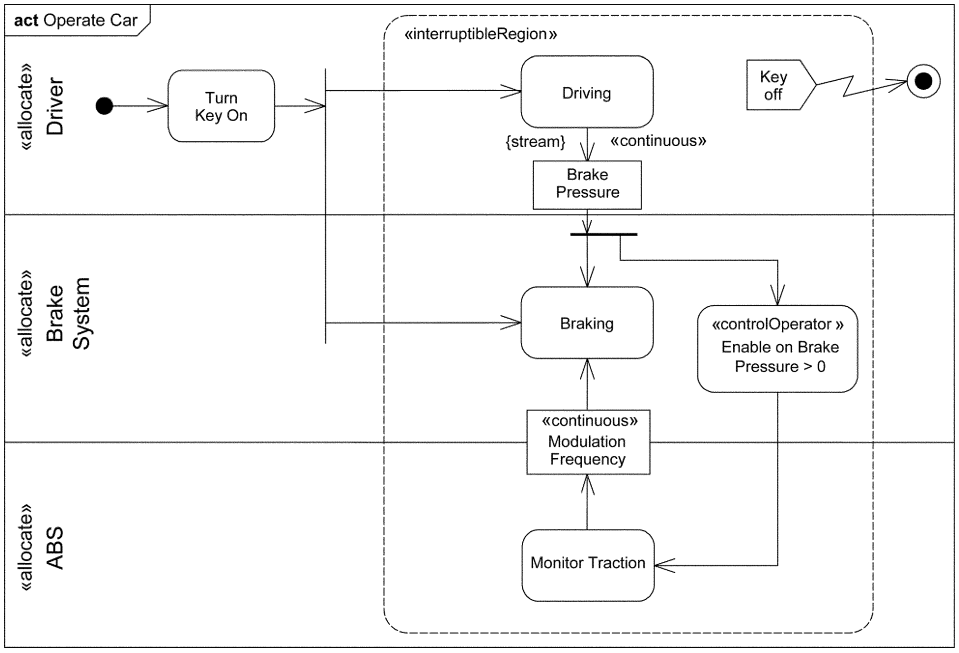


Figure 33. Activity with partitions.

added to indicate which aspect of the design structure is responsible for which activities. UML 2 requires that the activities in each partition are the responsibility only of the class represented by the partition. SysML provides the «allocate» stereotype, which loosens this restriction so that the activities in each partition are allocated to the partition class, but can also be allocated to other classes. UML 2 also supports multiple sets of partitions on a single diagram [Bock, 2004c], to show multiple allocations. SysML provides much more support for relating behavior and structure, for example, to allocate item flows in activities to assembly connectors. This is described in the SysML specification and will be covered in future articles.

7. REMAINING UML SE RFP REQUIREMENTS

The following UML SE RFP activity requirements remain to be filled by SysML and UML 2, due to lack of time to refine the requirements:

- *Stores*: The UML SE RFP calls for a model for storage of items as they move through an activity, where the store is persistent, and either depletable or nondepletable (6.5.2.1.2). The corresponding construct in UML 2, central buffer nodes [Bock 2004a], only hold objects while their containing activities are executing, not persistently across activity executions as in UML SE RFP stores or traditional data stores. When the activity is disabled, central buffer nodes no longer store any items, even though these items are not destroyed. Also, items flow out of UML 2 central buffer nodes as they become available, rather than as they are needed. UML 2 uses a form of item flow that treats items like a control value rather than storage. UML provides a limited form of non-depleting store that copies data as it leaves the store.

Persistent storage can be achieved with the predefined UML 2 actions for modifying persistent objects. Specifically, an item flow coming into a UML-SE store or traditional data store is equivalent to assigning that item or data to a particular place in storage. For example, the average pressure on a wing surface might be stored as the value of a particular attribute in an object that records information about the wing. Or water flowing into a physical system might be stored in a tank, which can be modeled as a dynamically changing characteristic of the tank. This is equivalent to a UML 2 action for writing attribute values. Conversely, an item flow going out from a UML-SE or traditional data store is equivalent to retrieving that item from a particular place in storage. In the previous example, the average pressure on the wing is read from an attribute of the wing object. Or water flowing out of a physical system might be taken out of a tank. This is equivalent to a

UML 2 action for reading and modifying attribute values. In this way a UML-SE store or traditional data store can be defined as an aggregate of primitive actions on persistent storage. There is currently no concise, standard graphical notation for this way of modeling stores.

- *Replication*: The UML SE RFP calls for support for activities that have multiple concurrent executions for a single usage, possibly determined by a control operator, with specification of the number of concurrent executions allowed (6.5.2.1.3 k, 6.5.2.2.2 c). The requirement has not been refined enough to determine how UML can support it. The UML constructs currently being considered are reentrant activities (Section 5.2.2), multiplicity constraints on activity decomposition (Section 6.1), and expansion regions [Bock 2005b].
- *Resources*: The UML SE RFP calls for specification of resources, which are generated, consumed, produced, and released when an activity executes (6.5.2.1.3). Resources do not appear on EFFBD or activity diagrams as inputs or outputs, even though they participate in the execution of functions and activities. UML supports preconditions and postconditions on activities and activity usages, but the effect on execution is not specified. Activities can be defined to proceed only when they have enough resources, but this requires modifications to the activity at each point in the flow, since resources may be exhausted at any time. An activity should support specification of various constraints on resources, such as those required to start, continue, and stop execution, as well as actions for claiming and releasing resources.

These requirements will be addressed in future versions of SysML or UML.

8. CONCLUSION

This article describes how SysML and the finalized UML 2 respond to the requirements of INCOSE and OMG for a systems modeling language based on UML. It organizes SysML and UML capabilities on a spectrum ranging from streaming to nonstreaming activities. It updates the mapping between EFFBD and UML 2 Activities given in an earlier article [Bock, 2003a], including one-to-one mappings as well as patterns and execution semantics for activities. It covers the additional capabilities of SysML and UML 2 that support streaming activities, which enhance control flow with some aspects of item flow, and provide for flow rate specifications, including continuous flow. It also describes decomposition and allocation of activities.

ACKNOWLEDGMENTS

The author thanks Sanford Friedenthal for many discussions and detailed reviews of this article, and his leadership in the SysML standardization effort. Thanks also to Carolyn Boettcher and Fredrick (Rick) Steiner for their contributions and perspectives. The input of James Long and Joseph Skipper was essential to developing an accurate translation between EFFBD and UML 2 Activities, and development of the SysML extensions.

REFERENCES

B. Blanchard and W. Fabrycky, System engineering and analysis, Prentice Hall, Englewood Cliffs, NJ, 1990.

C. Bock, UML 2 activity model support for systems engineering functional flow diagrams, J Int Council Syst Eng 6(4) (October 2003a), 249–265.

C. Bock, UML without pictures, IEEE Software Special Issue on Model-Driven Development 20(5) (September/October 2003b), 33–35.

C. Bock, UML 2 activity and action models, J Object Technol 2(4) (July/August 2003c), 43–53, http://www.jot.fm/issues/issue_2003_07/column3.

C. Bock, UML 2 activity and action models, Part 2: Actions, J Object Technol 2(5) (September/October 2003d), 41–56, http://www.jot.fm/issues/issue_2003_09/column4.

C. Bock, UML 2 activity and action models, Part 3: Control nodes, J Object Technol 2(6) (November/December 2003e), 7–23, http://www.jot.fm/issues/issue_2003_11/column1.

C. Bock, UML 2 activity and action models, Part 4: Object nodes, J Object Technol 3(1) (January/February 2004a), 27–41, http://www.jot.fm/issues/issue_2004_01/column3.

C. Bock, UML 2 composition model, J Object Technol 3(10) (November/December 2004b), 47–73, http://www.jot.fm/issues/issue_2004_011/column5.

C. Bock, UML 2 activity and action models, Part 5: Partitions, J Object Technol 3(7) (July/August 2004c), 37–56, http://www.jot.fm/issues/issue_2004_07/column4.

C. Bock, Inputs and outputs in the process specification language, Internal Report 7152, U.S. National Institute of Standards and Technology, Gaithersburg, MD, http://www.nist.gov/msidlibrary/doc/nistir7152.pdf, August 2004d.

C. Bock, Systems engineering in the product lifecycle, Int J Prod Dev 2(1–2) (2005a), 123–137.

C. Bock, UML 2 activity and action models, Part 6: Structured activities, J Object Technol 4(4) (May/June 2005b), 43–66, http://www.jot.fm/issues/issue_2005_05/column4.

S. Friedenthal and R. Burkhart, Extending UML from software to systems, INCOSE Symp, July 2003.

S. Friedenthal and C. Kobryn, Extending UML to support a systems modeling language, INCOSE Symp Proc, Toulouse, France, June 2004.

J. Grady, System requirements analysis, McGraw-Hill, New York, 1993.

E. Herzog and A. Torne, Support for representation of functional behaviour specifications in AP-233, Seventh IEEE Int Conf Workshop Eng Comput Based Syst, Edinburgh, April 3–7, 2000.

F. Kockler, T. Withers, J. Poodiack, and M. Gierman, Systems engineering management guide, 000802001202-5, Defense Systems Management College, U.S. Government Printing Office, Washington, DC, 1990.

J. D. Lambert, Numerical methods for ordinary differential systems: the initial value problem, Wiley, New York, 1991.

J. Long, Relationships between common graphical representations in system engineering, ViTech Corporation, Vienna, Virginia, 2002.

J. Long, M. Alford, M. Dyer, L. Marker, et al., The software requirements engineering methodology (SREM) notebook; TRW CDRL A006, BMDATC Contract DASG 60-75-C-0022. U.S. National Institute of Standards, Gaithersburg, MD, December 1975.

MathWorks, "Using Simulink®," http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/ sl_using.pdf, 2004.

D. Oliver, T. Kelliher, and J. Keegan, Jr., Engineering complex systems with models and objects, McGraw-Hill, New York, 1997.

OMG (Object Management Group), Report of the UML 2.0 Superstructure Finalization Task Force to the Platform Technical Committee of the Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/04-10-01, October 2004.

OMG (Object Management Group), UML 2.0 Superstructure Specification, http://www.omg.org/cgi-bin/doc?formal/05-07-04, July 2005.

OMG (Object Management Group), Model-driven architecture, http://www.omg.org/mda, 2006.

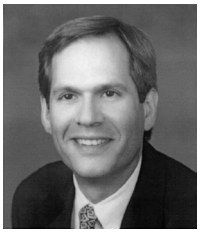
SE-DSIG (OMG Systems Engineering Domain Special Interest Group), UML for systems engineering RFI, http://www.omg.org/technology/documents/UML_for_Systems_Engineering_RFI.htm, February 1, 2002.

SE-DSIG (OMG Systems Engineering Domain Special Interest Group), UML for systems engineering RFP, http://www.omg.org/cgi-bin/doc?ad/03-03-41, March 2003.

SE-DSIG (OMG Systems Engineering Domain Special Interest Group), http://syseng.omg.org, 2005.

J. Skipper, private communication, 2005.

SysML Merge Team, Systems Modeling Language (SysML) Specification, http://www.omg.org/cgi-bin/doc?ad/06-02-01, February 2006.



Conrad Bock is a Computer Scientist at the National Institute of Standards and Technology specializing in process modeling and UML. He studied at Stanford, receiving a B.S. in Physics and an M.S. in Computer Science. His previous experience includes business process modeling at SAP and Microsoft. Mr. Bock leads efforts on process modeling in UML at the Object Management Group (OMG), and is contributing to the submission on UML for Systems Engineering to OMG.