Regular Paper

UML 2 Activity Model Support for Systems Engineering Functional Flow Diagrams*

Conrad Bock[†]

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

Received 22 June 2003; Accepted 18 July 2003
DOI 10.1002/sys.10053

ABSTRACT

This article compares Activity models of the Unified Modeling Language, version 2 (UML 2) [OMG (Object Management Group), UML 2.0 superstructure specification, August 2003, http://www.omg.org/cgi-bin/doc?ptc/03-08-02], to a widely-used systems engineering (SE) flow diagram, the Enhanced Functional Flow Block Diagram (EFFBD) [J. Long, Relationships between common graphical representations in system engineering, ViTech Corporation, 2002], and to the requirements for functional flow modeling in a systems engineering extension for UML (UML-SE) [OMG Systems Engineering Domain Special Interest Group (SE-DSIG), UML for systems engineering RFP, March 2003a, http://www.omg.org/cgi-bin/doc?ad/03-03-41]. Issues are identified in applying UML 2 Activities to EFFBD and to satisfying UML-SE functional flow requirements. Solutions are suggested to these issues that can be used to translate between the languages and to develop standards such as revisions to UML 2 or extensions in UML-SE. *© 2003 Wiley Periodicals, Inc. Syst Eng 6, 249–265, 2003

Key words: functional flow, UML, EFFBD, activity diagram

1. INTRODUCTION

Although systems engineering has existed as a discipline for several decades and been successfully applied to a wide range of complex products, it still lacks a standard modeling language. Organizations using multiple languages have less effective communication, increased project cost, and decreased product quality. Many of the other disciplines that systems engineering

*This manuscript is a U.S. Government work, an official contribution of the National Institute of Standards and Technology, and, as such, is not subject to copyright in the United States.

[†] E-mail: Conrad.bock@nist.gov.

Systems Engineering, Vol. 6, No. 4, 2003
© 2003 Wiley Periodicals, Inc.

interacts with have adopted standard modeling languages, most recently software.

To address this issue, the International Council on Systems Engineering (INCOSE) initiated an effort with the Object Management Group (OMG) to adapt UML for full-lifecycle systems engineering [Friedenthal, 2003]. A priority was set on aligning the underlying meaning of the adapted UML with traditional systems engineering models. This ensures that tools implementing the adapted UML can reliably interchange system designs with existing systems engineering tools. A second consideration is alignment of notation. It is beneficial for software and system engineers to communicate with the same diagrams; however, the legacy of existing notations in each community may delay this. With these goals, OMG's Systems Engineering Domain Special Interest Group (SE-DSIG) [SE-DSIG, 2003b] was formed and began by providing feedback to UML 2, the recent major revision of UML, to enhance its support for systems engineering.

The SE-DSIG also developed requirements for extending and adapting UML 2 for systems engineering (UML-SE), which were issued in March 2003 [SE-DSIG, 2003a]. These requirements are for extensions to UML 2 supporting the analysis, specification, design, and verification of a wide range of complex systems, which may include hardware, software, data, personnel, procedures, and facilities. They are to provide a complete, consistent, and standards-based representation of systems across the development lifecycle. The SE-DSIG also worked closely with the International Organization for Standardization's ISO 10303, informally known as the Standard for the Exchange of Product model data (STEP), in particular Application Protocol 233 for systems engineering (AP-233), to align the requirements of UML-SE with the evolving AP-233 neutral data interchange standard for systems engineering. Final submissions to satisfy the requirements are expected in 2004.

This article focuses on functional flow in systems engineering, as embodied in a widely-used systems engineering diagram, the Enhanced Functional Flow Block Diagram (EFFBD) [Long, 2002; Long et al., 1975; Skipper, 2003; Blanchard, 1990; Grady, 1993; Kockler, 1990; Oliver, Kelliher, and Keegan, 1997], and to the requirements for functional flow in UML-SE. The EFFBD, also called a behavior diagram, has been in use for three decades. Three tools have supported it over that time, and there is substantial heritage design information in existence based on this form of behavior modeling. EFFBD is also executable and produces timing information.

UML originated in the software community, and was intended to support software development, including representation of structure, behavior, and deployment of software systems. In addition, UML provides customization features that can be used when applying it to specific disciplines. As a result, it has been successfully applied in the systems engineering community, as indicated by the responses to the UML-SE Request For Information issued by the SE-DSIG [SE-DSIG, 2002].

UML also defines a repository for storing models that enables consistency maintenance between various views and analyses, connection between requirements and concrete entities that satisfy them, and support for generation of software and formats for automatic manufacturing. This flexibility includes multiple notations, so system engineers can employ familiar diagrams to manipulate and read the repository. It can be accessed dynamically as an information service, or serially though file-based interchange [Bock, 2003b].

UML 2 in particular introduces several new features important to systems engineering [OMG, 2003]. Hierarchical structure and behavior, component interconnection, and information flow are much more effectively addressed than in earlier versions. UML 1.5 and 2.0 also introduced models for parameterized functions defined or coordinated by control and data flow. For the first time UML supports functions that can be used without objects to host them, as in structured analysis, while maintaining object-orientation (OO) as an option. These facilitate application of UML to systems engineering by supporting traditional structured approaches, and incremental use of OO for allocating functions to system components in a flexible manner. The new features considerably widen potential applications of UML.

UML 2 defines a model and execution machinery based on flow modeling intuition, called *Activities*, and a set of predefined functions for primitive actions such as object creation [Bock, 2003a]. These support systems engineering applications with enhanced data and control flow constructs and multidimensional partitions that can represent a component, among other features, such as cycles and queuing. UML Activities and State Machines have been completely executable since UML 1.5 and have always been partially executable due to their definition of a virtual machine.

Consequently, UML 2 Activities support flow modeling across a wide variety of domains, from computational to physical. This makes them ideal for specifying systems independently of whether the implementation is software or hardware, and independently of where the system/environment boundary is drawn. Finally, the combination of Activities and Actions retains the UML

1.x capability of reacting to events, so they can be applied to areas requiring that, such as embedded systems.

Before UML 2, object-oriented software engineering did not usually address flow models very well,[1] so there is little existing literature that compares flow modeling in software and systems engineering. Work on UML and systems engineering that discusses functional flow does not address the UML 2 flow models [Ögren, 2000; Pandikow and Törne, 2001a]. Other literature on systems engineering and UML does not address functional flow [Axelsson, 2002; Bahill and Daniels, 2003; Lykins, Friedenthal, and Meilich, 2000]. There is a proposed framework for mapping metamodels of software and systems engineering models, but not the actual mapping [Pandikow and Törne, 2001b].

This article contributes to the efforts on bridging software and systems engineering by comparing the semantics of flow models of UML 2 and systems engineering. The results can be used to facilitate the integration of software and systems modeling by translating between the domains and developing standards supporting such integration, such as revisions to UML 2 or extensions in UML-SE.

The next section gives a detailed comparison of UML 2 Activities with EFFBD and with UML-SE requirements for function-based behavior.[2] Each subsection describes the aspects of EFFBD and UML-SE that are supported by UML 2 Activities, and aspects where EFFBD and UML-SE are not directly supported by UML 2. EFFBD and UML-SE are described together when they have the same characteristics, and separately when they are different. Only EFFBD diagrams are used in figures, because the notation of UML-SE has not been determined yet.

Suggested solutions for alignment issues are given in Section 3. These may be due to discrepancies between EFFBD and UML 2, between UML-SE and UML 2, or both. Solutions are proposed as changes to UML 2 or extensions to UML 2 in UML-SE.

[1]A notable exception is James Odell's Object-oriented Information Engineering [Martin and Odell, 1992].

[2]UML provides other behavior models, namely, interactions, which focus on messages between objects, and state machines, which cover object states and transitions between them. This paper focuses on Activities because they are the closest UML behavior model to SE functional flow diagrams, providing for sequencing of function execution by both control and data.

2. COMPARISON OF UML 2 ACTIVITIES WITH FUNCTIONAL FLOW BLOCK DIAGRAMS AND UML-SE FUNCTIONAL FLOW REQUIREMENTS

Figures 1 and 2 show an example EFFBD that will be referenced in the comparison, with one of the elements in Figure 1 decomposed into the diagram in Figure 2. Figure 3 gives the corresponding UML 2 Activity Diagram. Both EFFBD and Activity Diagrams give the sequence and conditions for execution of functions. For example, function 2.3 in Figure 2 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 2, function 2.2 takes input of type Item 1 from the output of function 2.1. Function 2.2 cannot start until Item 1 arrives. The details of EFFBD and Activity Diagram execution are covered in the subsections below for each construct.

The UML-SE requirements are designed to support EFFBD, UML 2 Activities, and additional capabilities to provide a comprehensive model for functional flow in systems engineering applications. For example, UML-SE requirements have additional flexibility in control and data flow compared to EFFBD or UML2, such as control flow for disabling a function as well as enabling it, and treating control as a form of input and output. The details of UML-SE execution are covered in the subsections for each construct below.

Table I shows the correspondence between constructs in the EFFBD, Activity Diagram, and the UML-SE requirements that are discussed in each subsection. There are three kinds of construct[3]:

1. Functional: These describe basic behaviors or transformations that are coordinated in a functional flow diagram.
2. Data/object/item flow: These are how a flow diagram routes data, objects, and other items, such as energy and matter, between functions.
3. Control: These are how a flow diagram starts and stops execution of functions.

For clarity, this paper introduces two terms for concepts that are implicit in EFFBD and UML-SE but not named (*function usage* and *function port usage*). These are listed in the table next to the equivalent UML 2 terminology.

[3]Some of the constructs connect categories. For example, inputs and outputs to a function (function ports in UML-SE) are defined by the function itself, not the data/object flows that use the function, and so are classified under functions. However, data/object flows must use inputs and outputs, and do so through function port usages. Likewise, multiexit functions are technically a kind of function, but affect control flow choices in the diagrams that use them. These are classified under control.

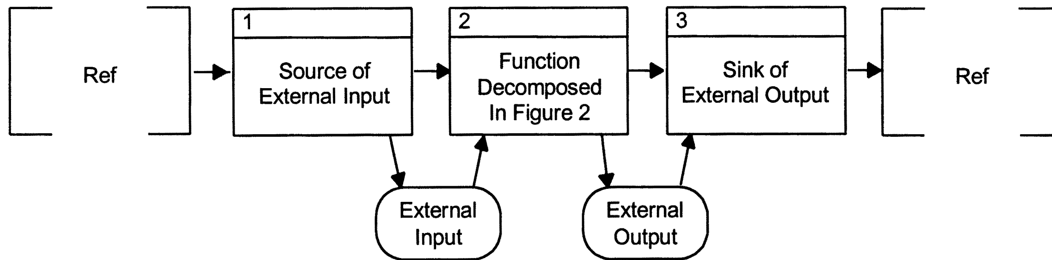

Figure 1. Enhanced functional flow block diagram context for Figure 2.

2.1. Activation ⇔ Execution

A UML-SE activation refers to the running of a function or behavior in a working system (requirement 6.5.2.2). It is called an *execution* in UML 2. Activations and executions are useful in explaining what a behavior diagram does, even though they do not actually appear on a diagram. UML-SE requires that the semantics of flow diagrams be defined in terms of execution rules (6.5.2.3), which is the same approach as in UML. UML-SE also requires a way to specify how long a function is active and inactive (6.5.2.4.1). The UML 2 time model supports specifying constraints on time durations.

2.2. Function (or Activity) ⇔ Behavior

An EFFBD or UML-SE function is a reusable definition of a transformation that accepts inputs and provides outputs, which can include modifications to objects. A function has a name and defines the types of entities that may be accepted as inputs and provided as outputs (requirement 6.5.2.1.3 a; see Section 2.4). In UML-SE, functions are required for creation, destruction, monitoring, and modification of elements, and a null transformation (requirement 6.5.2.1.3 b).[4] Functions can be decomposed into a complete diagram representing the next level of hierarchical behavior, as Function 2 is in Figures 1 and 2 (requirement 6.5.2.1.3 g; see Section 2.4). In UML-SE functions may be defined by mathematical expressions (requirement 6.5.2.1.3 i). UML-SE provides for allocating behaviors, including functions, to systems (requirement 6.5.2.5). This means the system performs the function.

In UML 2, these are called *behaviors* and provide the features above. The inputs and outputs of a behavior are called *parameters* (see Section 2.4). They define the type of entity that is input or output, which are called *classifiers* in UML. UML 2 behaviors can be allocated to any classifier, including a component or system. This can be shown in an activity diagram using swimlanes (partitions) that represent components or systems. Swimlanes divide the diagram into sections. Any usage of a function within the swimlane is allocated to the component or system it represents.

Functions and behaviors are usually shown either in the diagram in which they are used, as Function 2 is in Figure 2 (see Section 2.3), or they are decomposed into a complete diagram as in Figure 3.

Three UML-SE requirements on functions are issues to address in applying UML 2:

- Functions require resources, which are generated, consumed, produced, and released when the function executes (requirement 6.5.2.1.3 e).

The effect of behaviors in UML 2 is specified at each usage of behavior (action; see Section 2.3), and is limited to effects on the entities that are input or output. This allows each usage to specify a different effect, depending on context, but is more cumbersome if the effects are the same for all uses of a particular behavior.

- Functions may be interruptible or not (requirement 6.5.2.1.3 h).

UML 2 does not indicate whether a behavior is interruptible.

- The number of replicated functions that can concurrently execute (requirement 6.5.2.1.3 k) is also specified.

The requirements for replication in EFFBD and UML-SE are still open; however, they will relate to Iteration and Loop (see below).

[4]Creation and destruction of objects and items does not apply to conserved entities such as energy. These can only be transformed, not destroyed.

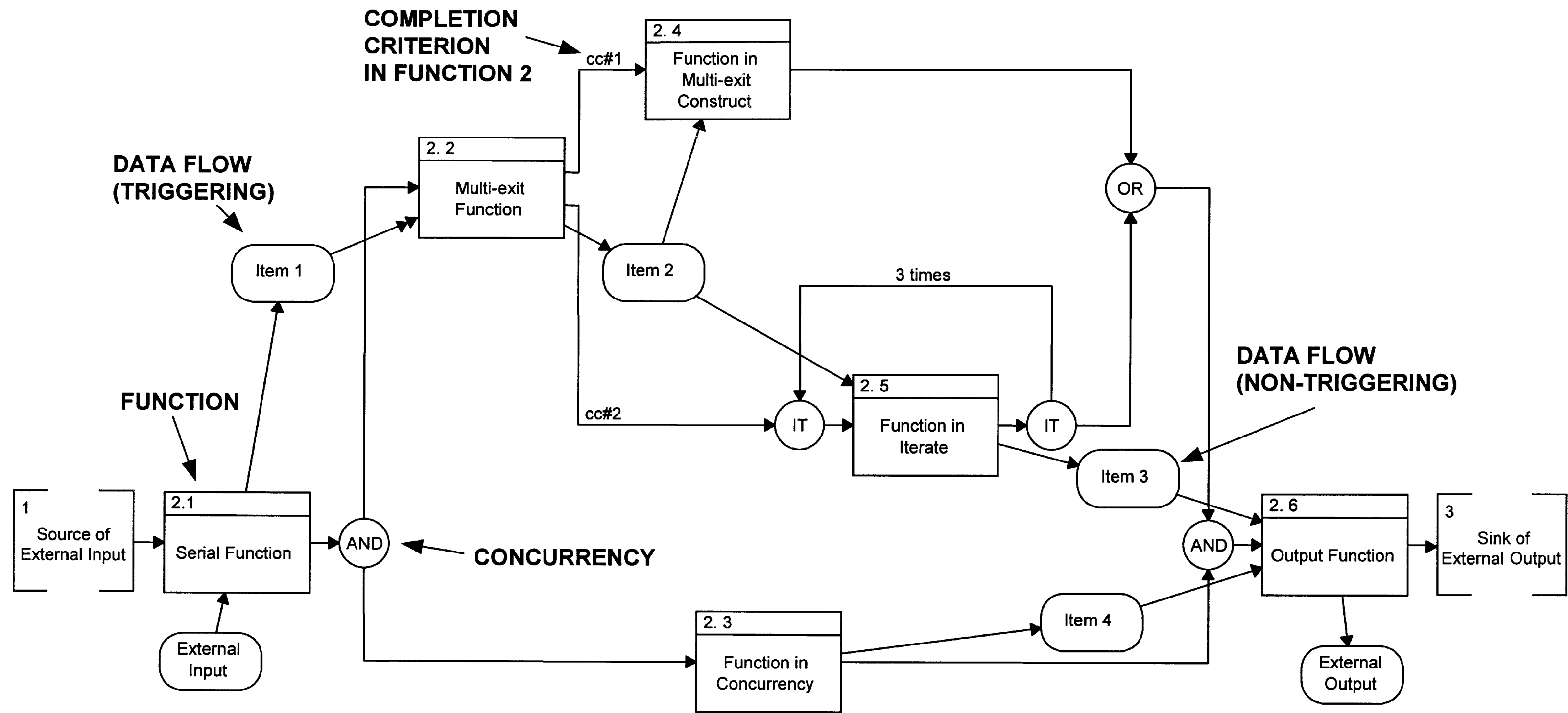

Figure 2. Enhanced functional flow block diagram for function 2 in Figure 1.

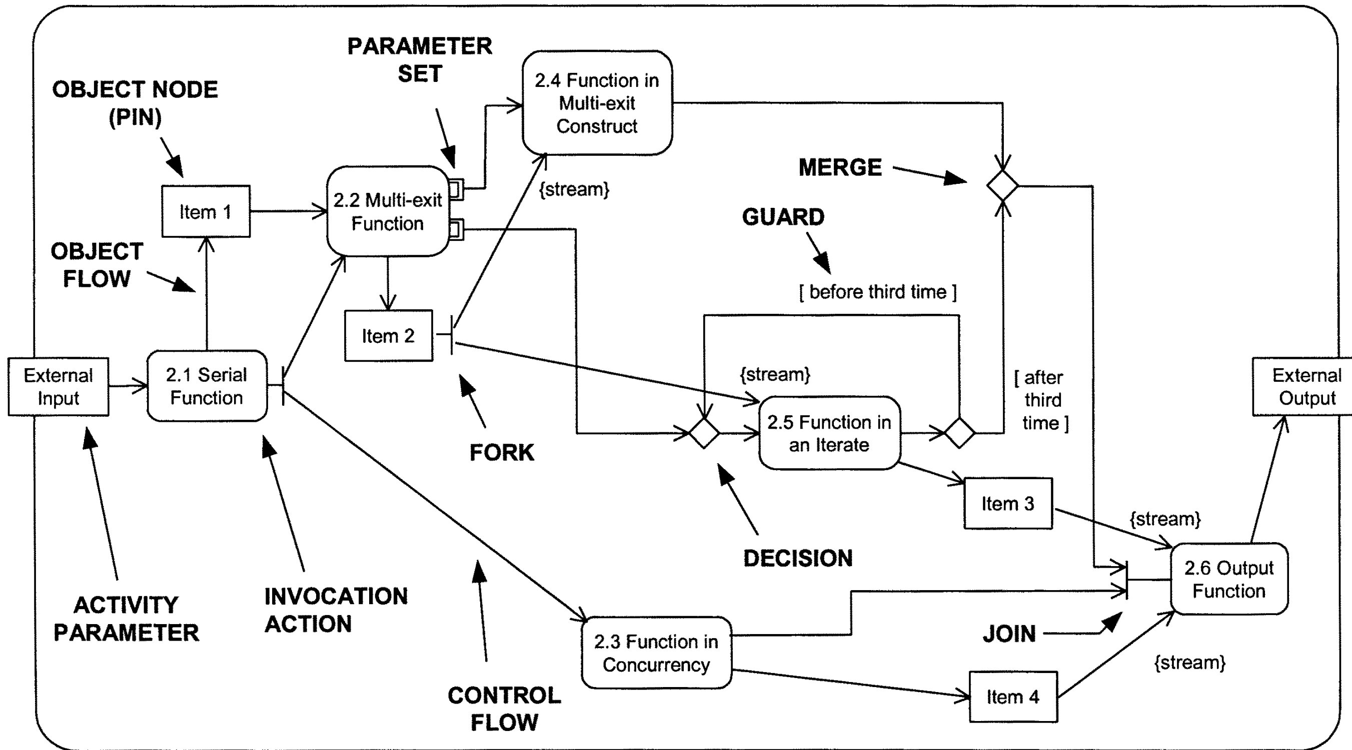

Figure 3. UML 2 Activity diagram corresponding to Figure 2.

Table I. Correspondence between EFFBD Constructs, UML 2 Activity Constructs, and UML-SE Requirements

	EFFBD	UML 2 Activity	UML-SE Requirement
Function		Execution	Activation, 6.5.2.2, 6.5.2.3, 6.5.2.4.1
	Function (or Activity)	Behavior	6.5.2.1.3 a-b,e,g-i,k
		Action (Function Usage)	
	External Input/Output	Activity Parameter Node	Function Port: 6.5.2.1.1, 6.5.2.1.3 c,d
Data/ Object/ Item Flow	Item Flow	Object Flow	6.5.2.1.3 f,g,j
	Item Node	Pin (Function Port Usage)	
	Triggering Item Input	Nonstreaming Parameter	Special case of 6.5.2.2.2, 6.5.2.2.3
	Non-triggering Item Input	Streaming Parameter (issues)	Special case of 6.5.2.2.2, 6.5.2.2.3
		Data Store Node (with issues)	System Store, 6.5.2.1.2
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 (with issues)	Special case of 6.5.2.2.2, 6.5.2.2.3
	Completion Criteria	Postconditions on parameter sets (with issues)	Special case of 6.5.2.2.2, 6.5.2.2.3
	Iteration, Loop	Flow, Decision, Merge	6.5.2.2.2 c
		Join with Join Expression (with issues)	Control Operator, Activation Events: 6.5.2.2.2, 6.5.2.2.3

2.3. (Function Usage) ⇔ Action

EFFBD and UML-SE diagrams use functions without affecting the internal aspects of those functions (requirement 6.5.2.1.3 c). This is so the same function can be used multiple times in many diagrams, without constraining how the diagram uses the function. For example, function 2.2 in Figure 2 happens to be used after function 2.1 in that particular diagram, but other EFFBDs could reverse the order by getting Item 1 from another source. The term *function usage* is introduced in this paper to distinguish between a function and where it appears in diagrams. It is notated in EFFBD as a rectangle, as shown in Figure 4 below, or the rectangles in Figure 2.

In UML 2, these are called *actions* and provide the features above. Actions are a point in the flow of an Activity that executes a behavior. They are notated as

Figure 4. EFFBD function usage, UML 2 action.

round-cornered rectangles. It can be thought of as a behavior usage. UML 2 predefines some actions for primitive capability such as creating objects, setting attributes, and so on.[5]

2.4. External Input/Output, Function Port ⇔ Activity Parameter Node (a Kind of Object Node)

Inputs and outputs to a function define what types of entities will be provided to the function and what types of entities will be the result of the function (6.5.2.1.1 a, c). They are defined separately from flows (Section 2.5) so that a function is guaranteed to be provided the same inputs and expected to provide the same outputs regardless of what flow diagrams use it. Sometimes the terms *input* and *output* are used in short to mean the type of entity being input or output. For example, if a system takes water as input, then the water is called an input. However, water per se is not an input, it is an input only because a system takes it as such.

The inputs and outputs of a function consist of a name and the type of entity that is input or output. For example, a system may take an input named coolant where the type of entity being input is water. These two aspects can be changed independently, for example, the type of the coolant input could be changed to air.

UML-SE introduces the term *function port* to represent these two aspects of inputs and outputs (requirements 6.5.2.1.1 c, 6.5.2.1.3 c). It binds a type of entity to a function that transforms it. EFFBD uses the terms *external input/output*, though this can also mean the type of entity that is flowing. Entities that are input and output can be decomposed into constituent parts and have properties that vary continuously over time (requirement 6.5.2.1.1 a, b). EFFBD inputs and outputs are notated as a round-cornered rectangle at the beginning and end of the flow. See Figure 5 below and the nodes labeled External Input and External Output in Figure 2.

[5]These might be considered behaviors, but are modeled directly as actions in UML 2. Actions achieve reusability by making a new instance of an action class for each usage in a flow [Bock, 2003a].

In UML 2, the inputs and outputs to activities are called *parameters* and provide the features above. They are notated as rectangles on the boundary of the Activity, as shown at the bottom of Figure 5.

The following UML-SE requirement on function ports is an issue to address in applying UML 2:

- Functions specify how input and output is handled, including how inputs are queued, stored, discarded, and how they may interrupt an active function (requirement 6.5.2.1.3 d).

The queuing of inputs in UML 2 is specified for each usage of a behavior in an Activity (action; see Section 2.3), rather than on behaviors themselves. This allows each usage to have different queuing rules, but is cumbersome if these rules are the same for all uses of a particular behavior.

UML 2 does not provide explicit specification of how inputs are stored, discarded, or if they interrupt an action that is already active. Inputs arriving at an action that is already active will either start a new execution of the action in parallel with the existing one, or be queued until the action finishes, depending on whether the behavior is reentrant or not.

2.5. Item Flow ⇔ Object Flow; Item Node, Function Port Usage ⇔ Pin; Triggering Data Input ⇔ Nonstreaming Parameter; Nontriggering Data Input ⇔ Streaming Parameter (Issues)

A data or item flow in an EFFBD or UML-SE requirements specifies how the output of one function is passed to the input of another (6.5.2.1.3 f). Data/item flows connect function usages with what this article calls *item nodes*. The notation for item nodes is a round-cornered rectangle. For example, Item 1 in Figure 2 is an item node. Item flows can also pass the external input of an EFFBD or UML-SE flow diagram to the input of a contained function usage, and pass the output of a function usage to the external output of the containing diagram (requirement 6.5.2.1.3 g). For example, Figure 2 shows an external input being passed to function 2.1 and function 2.6 passing items to an external output.

Since functions can be reused in many diagrams, the flows to and from a function are mediated through what this article calls *function port usages*, which tie the flows into and out of a function usage to the input and output ports of the function itself. This allows a function to be used in multiple flows without one flow affecting another. The item flow and item node notation omit the ports on the function usage they connect, for concise-

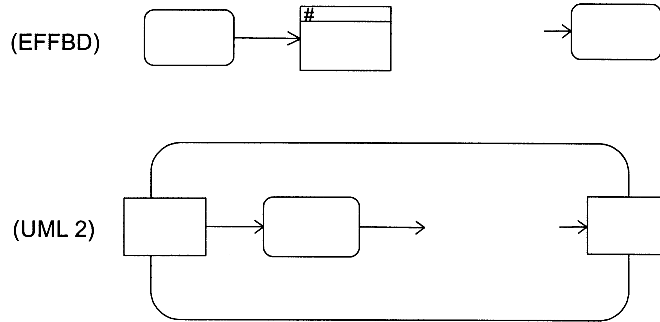

Figure 5. EFFBD external input and output/function port, UML 2 Activity parameters.

ness. For example, the usage of function 2.1 in Figure 2 has a function port usage outputting a value of type Item 1 to an input port for the usage of function 2.2.

Item inputs to an EFFBD function usage are called *triggering* if they are required to arrive at the usage before the function can be activated. For example Item 1 in Figure 2 is required to arrive for function 2.2 to be activated. Item inputs arriving at an EFFBD function usage that is already activated are queued until the activation is finished. Triggering item inputs are notated with an item node that has a double-headed arrow pointing to the function taking the input. Triggering items in UML-SE is a special case of a control operator (see Section 2.12).

UML 2 object flow constructs correspond to EFFBD and UML-SE constructs as given in the heading (Item Flow ⇔ Object Flow, etc.), and support the features above, except for issues with nontriggering inputs (see below). If a nonstreaming input arrives at an action that is already executing, it can begin a new execution of the behavior if the behavior executing is reentrant; otherwise it is queued. The notation for UML 2 Object Node is a rectangle, labeled with the type of flowing object. The rectangle can be shown in the middle of an object flow, or as two small rectangles for the input and outputs of actions, which are called *pins* (Figs. 6–9 below).

Three aspects of EFFBD and UML-SE are issues to address when applying UML 2:

- Multiple flows coming out of an EFFBD item node are equivalent to a UML 2 object node with a fork for the outgoing flows.

For example, Item 2 in Figure 2 has a fork after it to provide the items to two functions. This is because UML 2 pins do not copy values that flow out of them as EFFBD item nodes do (compare to Section 2.6).

- Nontriggering item inputs in EFFBD are not required for activation, but are used if they are available when activation starts.

For example, Item 3 in Figure 2 is not required for function 2.6 to be activated. Nontriggering item inputs are notated with an item node that has a single-headed arrow pointing to the function taking the input. Nontriggering item in UML-SE is a special case of a control operator (see Section 2.12).

UML 2 streaming inputs are not required to start execution, though they are required to arrive before the execution terminates. If a streaming input arrives at an action that is already executing, then the input will be accepted by the executing behavior.

EFFBD nontriggering inputs do not quite translate to UML 2 streaming inputs, because EFFBD will queue nontriggering items when it arrives at a function that is already activated, whereas UML 2 streaming inputs are

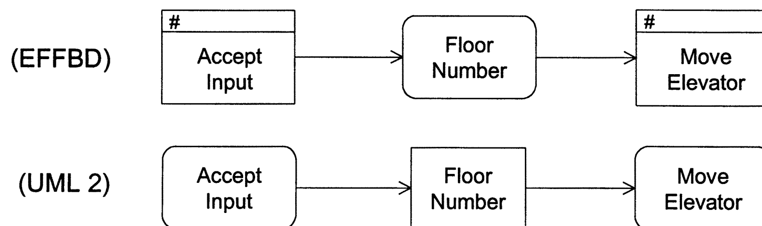

Figure 6. EFFBD item flow and UML 2 object flow.

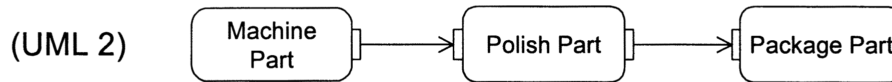

Figure 7. UML 2 pins (a kind of object node).

accepted by the executing behavior. UML streaming inputs are also required to arrive before the function is finished executing.

- UML-SE flow diagrams are required to support functions that transform continuous time varying inputs into continuous time varying outputs (requirement 6.5.2.1.3 j).

UML 2 defines a discrete virtual machine. This does not preclude modeling continuous transformations, but does not guarantee that values arriving at the same time at the same action are used by the same behavior execution.

2.6. System Store ⇔ Data Store Node (a Kind of Object Node, with Issues)

UML-SE requires a persistent store that may be depletable or not (6.5.2.1.2). For example, a water tank is a depletable store, whereas a database is a nondepletable store.

UML 2 defines two kinds of Object Node that temporarily store objects in either a depletable or nondepletable fashion. See issues below.

UML 2 Object Nodes only hold objects while their containing Activity is executing. In particular, the storage of objects or data does not affect persistent structure, such as attribute values of objects in the system. In addition, objects and data flow out of UML 2 Object Nodes as they become available, rather than as they are needed. Items in a UML-SE store may flow out when they needed or when they are available.

2.7. Control Flow ⇔ Control Flow

A control flow in an EFFBD or UML-SE requirements specifies sequential activation constraints. Control flow connects function usages to indicate that the target function can only start after the source function is finished (requirement 6.5.2.2.1 c, e–g, see issues below regarding deactivation control). For example, function 2.5 in Figure 2 starts after function 2.2 is finished. UML-SE function usages with no incoming control flows behave as if they always have their control input enabled (requirement 6.5.2.2.1 f).

UML 2 control flow supports the features above. It is shown in the lower part of Figure 10.

Three UML-SE requirements on control flow are issues to address in applying UML 2:

- Control flow in UML-SE can connect to or from a control function, which is a kind of function that processes control values as if they were data (see Section 2.12).
- EFFBD and UML-SE control can be queued until the function completes (requirement 6.5.2.2.1 a with 6.5.2.1.3 d).

Control flows in UML 2 are discarded if they arrive at an action that is already executing.

- UML-SE control flow can abort an existing activation of a function. Control values can be either enable or disable (requirement 6.5.2.2.1 b, d). There is a requirement to support timeouts, which also abort an existing activation (requirement 6.5.2.2.1 e).

Control flow in UML 2 only provides for enabling a new execution, but combinations of other constructs produce the same functionality.

2.8. Select ⇔ Decision. Merge Branch Annotation ⇔ Guard

A branch in the flow of an EFFBD or UML-SE flow diagram in which only one branch is taken at any particular time is called a *Select*. In UML-SE, they are

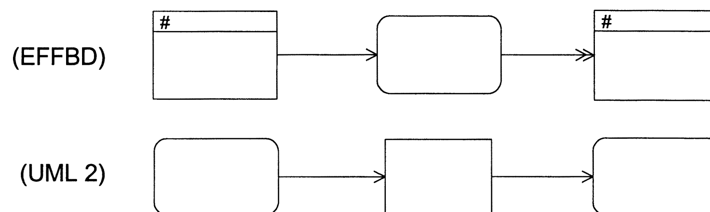

Figure 8. EFFBD triggering item flow and UML 2 nonstreaming object flow.

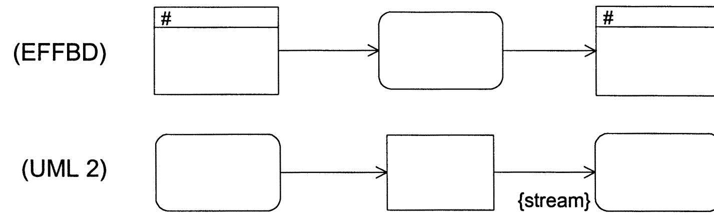

Figure 9. EFFBD nontriggering item flow and UML 2 streaming object flow.

a special case of a control operator (6.5.2.2.2 c; see Section 2.12). In EFFBD, Selects are shown with a small circle labeled "OR," (see Fig. 11). A point in the flow where selected flows come together is notated the same way, which in this article is called an *End Select*. For example, the circled OR in Figure 2 on the right side merges flows from function 2.4 and the end of the iteration over function 2.5. It has not been determined whether an End Select can be used with incoming flows that are not exclusive.

Flows coming out of an EFFBD Select are marked with branch annotations (see Fig. 11). These give conditions for proceeding along the branch they annotate.

UML 2 Decision and Merge support the features of Select and End Select, respectively. They are notated as diamonds (see Fig. 11). Merges can be used with incoming flows that are not exclusive; that is, any incoming flow during execution will always be passed to the outgoing flow of the merge, regardless of whether other flows have arrived before.

UML 2 Guards support the features of branch annotations. They are used on flows coming out of decision points. They give specific conditions for proceeding along the flow they are attached to.

One aspect of EFFBD is an issue to address when applying UML 2:

- EFFBD provides for stochastic choices at Selects. Branch annotations may give probability distributions for going along one flow or another, rather than specific instructions. UML 2 Guards do not provide for expressing probability distributions for decisions.

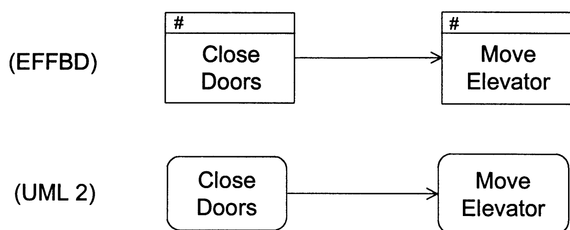

Figure 10. EFFBD and UML 2 control flow.

2.9. Concurrency ⇔ Fork, Join

A branch in a flow where the branches operate concurrently is called *Concurrency* in EFFBD and Fork in UML-SE (6.5.2.2.2 c). In UML-SE, they are a special case of a control operator (see Section 2.12). In EFFBD, Concurrency is shown with a small circle labeled AND, as shown after function 2.1 in Figure 2. A point where concurrent flows come together is notated the same way, and is called an *End Concurrency* in this article, as shown before function 2.6 in Figure 2.

UML 2 Fork and Join support the features of Concurrency and End Concurrency, respectively. See Figure 12.

2.10. Multiexit Function ⇔ Behavior Using ParameterSets (with Issues); Completion Criteria ⇔ Postconditions on ParameterSets (with Issues)

EFFBD multiexit functions have more than one control output, but for each activation only one control output is provided. In UML-SE, they are a special case of control operators (6.5.2.2.2 c; see Section 2.12). In the EFFBD example, function 2.2 in Figure 2 is multiexit.

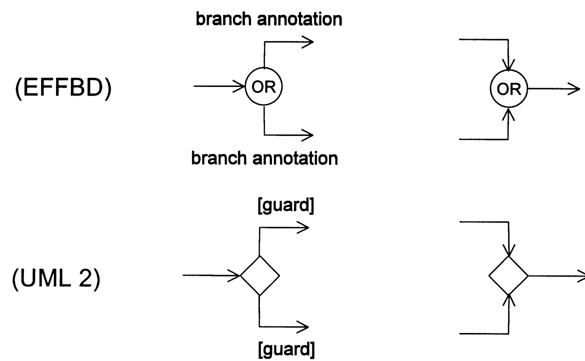

Figure 11. EFFBD select and branch annotation, UML 2 decision, guard, and merge.

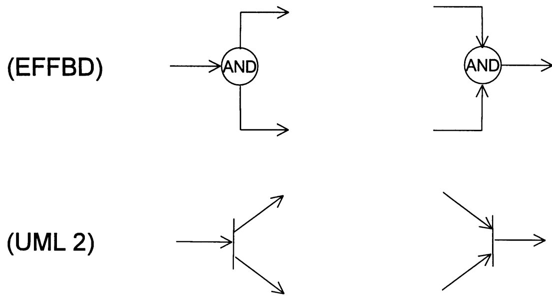

Figure 12. EFFBD concurrency, UML 2 fork and join.

The conditions that determine which output will be provided are specified internally to the function. These completion criteria are notated as annotations on the flows coming out of multiexit functions. It is an open question of whether multiexit functions can provide alternative item outputs.

UML 2 parameter sets are a way of grouping parameters of a behavior into sets such that each execution of the behavior provides inputs or outputs for the parameters in only one of the sets. See issues below for correspondence to multiexit functions. They are shown as boxes around pins, as shown on function 2.2 in Figure 3 and the lower part of Figure 13.

Two aspects of EFFBD multiexit are issues to address in applying UML 2:

- EFFBD multiexit is for control flow and UML 2 parameter sets are for data flow.
- EFFBD completion conditions require UML 2 to have postconditions on parameter sets, but postconditions are only supported on parameters currently.

2.11. Iteration, Loop ⇔ Flow, Decision, Merge

EFFBD Iteration and Loop are special nodes used to form cycles in the flow graph. They are used at the beginning and end of a cycle. In UML-SE, they are a special case of a control operator (6.5.2.2.2 c; see Section 2.12). EFFBD Iteration is notated with a circle labeled with IT, Loop with a circle labeled with LP. For example, the circles before and after function 2.5 in Figure 2 indicate iteration over function 2.5.

EFFBD Iteration determines the number of loops after the first cycle at runtime. The determination may be a constant or an expression evaluated at runtime. The determining expression is recorded on the iteration node at the end of the cycle, and notated on the flow that returns to the beginning of the cycle. EFFBD Loop determines when to stop the cycle each time through at the end of each cycle. The determination is modeled the same way as for iteration. EFFBD Replication is a similar construct that supports multiple activations of the same function usage. The details of replication are not completely determined yet.

UML 2 supports cycles by using a Merge at the beginning of the cycle and a Decision at the end (see the lower part of Fig. 14).The test for returning to the beginning of the cycle is on the guards coming out of the decision, which are notated with brackets annotating the flow line. See the flow line returning to the merge before function 2.5 in Figure 2. The same UML 2 action can be executed more than once, if it is invoking a reentrant behavior.

Two aspects of EFFBD and UML-SE cycles are issues to address in applying UML 2:

- UML 2 does not explicitly distinguish EFFBD loops and iterations.
- A UML-SE requirement provides for limiting the number of concurrently executing activations of a function (requirement 6.5.2.1.3 k). A UML 2 reentrant behavior can have an unlimited number of concurrent executions. Other behaviors can only have one.

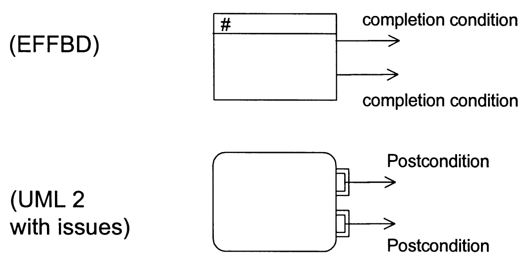

Figure 13. EFFBD multiexit function, UML 2 parameter sets (with issues).

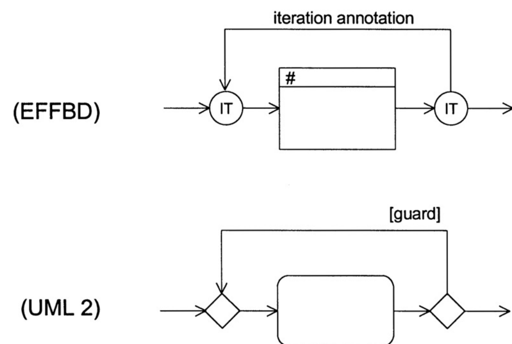

Figure 14. EFFBD iteration, UML 2 merge and decision.

2.12. Control Operator, Activation/Deactivation Events ⇔ Join with Join Expression (with Issues)

UML-SE requires a kind of function called a *control operator* that takes as input activation/deactivation events and other events, to combine them into a single control value to activate or deactivate a function (6.5.2.2.1 g, 6.5.2.2.2, 6.5.2.2.3). This can occur at any time based on a set of triggering events and conditions, and does not necessarily wait for another function to complete. The specific control constructs for selection, fork, and so on, are examples of control operators. A function usage can have a control operator to transform multiple incoming control flows, to determine how these are combined to affect the activation (requirement 6.5.2.2.1 a).

UML 2 control flows detect the completion of action executions and these can be combined with each other and with data flow using an expression on the Join construct. UML 2 provides for events as they affect objects: change events, signal arrival, operation calls, and time, time being the only global event. UML 2 actions with multiple incoming control flows must wait for control to arrive on all incoming flows before beginning execution. Actions do not provide for other combinations of control. See issues below.

One approach to applying UML 2 to control functions is to use join expressions. This aspect of UML 2 joins can represent any combination of control and data. However, control flow is limited to detecting completion of action execution and the invocation of operations on objects. Interrupted executions and newly started executions are not detected in UML 2 control flow (see Section 2.7). Also the combination of control and data flow in a join always produces a data flow. Joins cannot be used as the input combiner inside of an action.

Another approach is to extend UML behavior to manipulate control values directly as if they were data, and provide these behaviors as input combiners for actions. See Section 3.1.

3. CHANGES AND EXTENSIONS TO UML 2 ACTIVITIES FOR EFFBD AND UML-SE FLOW DIAGRAMS

This section proposes solutions to the issues raised in Section 2 comparing UML 2 Activities with EFFBD and UML-SE flow diagram requirements. The issues fall into three categories:

1. Parity between the capabilities of control and data flow, for example, queuing of control
2. Local versus global specification, for example, queuing specified at function usages or at functions
3. Other extensions to address continuous time varying inputs and outputs.

The suggested solutions below can be made as an extension in UML-SE or more generically to UML 2. The proposed approach is indicated in some cases below.

3.1. Parity of Control and Data

Some of the features of data flow in EFFBD, UML-SE, and UML 2 flow diagrams can apply to control, and vice versa:

- Control queuing: EFFBD and UML-SE flow diagrams can queue enabling control values that arrive at a function usage that is already activated, whereas UML 2 discards them (Section 2.7).

UML 2 pins can be extended to support control tokens by defining a special type for control values. Making control a form of input is more general, since it resolves the next two items also. See “Control as input and output” at the last bullet below.

- Multiple control values: UML-SE flow diagrams provide for two control values, enable and disable, whereas UML 2 only provides enable (Section 2.7).

UML 2 has other constructs for halting execution, for example, interruptible regions, and exception parameters. These disable actions executing in a region of the flow graph when particular events occur, and may be more suitable than disabling control values for some applications.

Assuming additional control values are needed in system engineering applications, it may be that more than enabling and disabling would be useful, such as suspend, resume, soft disable that allows cleanup behaviors, hard disable that has no cleanup behaviors, and so on. In effect, control values can become almost as varied as data. However many values are needed, the semantics of UML 2 control tokens can be extended to cover them, because they are simple additions to the existing enabling control value.

It is not determined yet how UML-SE will satisfy the requirement for multiple control values. For example, will control flow lines be marked as dedicated for enablement or disablement, so modelers can visualize them easily, or will flow lines carry either kind depending on runtime dynamics for flexibility? If other control values are introduced, such as aborts due to exceptional conditions, how will a function usage specify which

type of control is to be emitted when it completes? One possibility is the general solution given below, which is already required for other features.

It is also not determined how joins should behave when receiving different control values. Multiple enabling control values arriving at a join are normally combined into a single enabling control. Any two control values of the same type can be combined this way, but what about combining different values? For example, how are enabling and disabling control values combined at a join? Perhaps in this case one might say that they nullify each other, but it is unclear what principle is, and consequently cannot be generalized to other control values that might be introduced later, such as suspend, soft disable, and so on.

- Multiexit and parameter sets: EFFBD provides multiexit for control outputs only, whereas UML 2 does the same for data only (Section 2.10).

Both of these capabilities can be provided by making control a form of output (see the next bullet). EFFBD multiexit completion conditions translate to post-conditions on parameter sets, which UML 2 does not support currently. This feature should be a standard part of UML 2.

- Control as input and output: UML-SE control operators require control values as inputs and outputs (Section 2.12). Providing this capability to all functions resolves the above issues related to input and output. Control and data both can be typed, queued, participate in multiexit, and have specific values. Some other issues arise, however:
  - Control functions operate on control values, but are not affected by them. For example, a disablement control arriving at an already activated control function does not disable it. A simple way to distinguish this case is to allow function usages to have input port usages (pins in UML 2) that are not bound to function ports (parameters in UML 2). Port usages without ports have nowhere to pass the control value, so the values can be directed to whatever agent is executing the function.
  - UML 2 provides for data arriving at an already executing function usage and being consumed by that execution (streaming inputs and outputs). For control operators implemented as Activity models, these can be consumed by the execution by putting multiple control tokens at the initial node at different times within the same execution of the diagram.
  - Symmetry might imply that control disablement will also be queued when arriving at a function usage that is already disabled. It is not clear how useful this is, because the queued disabling control would only come into effect when the function is reactivated, immediately deactivating it. Likewise, any other control values that might be used, such as suspend, would be queued and present the issue. The proper application of queued control values is yet to be determined.

UML-SE function-based behavior requirements continue a long-term trend of unifying data and control. Modern flow models treat data as a form of control, by requiring data to be available for a function to activate, as compared to the traditional data store that passively provides information on request from already activated functions. UML-SE requirements likewise treat control as a form of data by providing multiple control values, control queuing, and control functions.[6]

Other aspects of control and data flow needing alignment between EFFBD and UML-SE requirements and UML 2 Activities are:

- UML-SE function usages with multiple incoming control flows can specify a control operator to determine how these are combined to affect the activation (Section 2.7). In UML 2 this is always "and."

UML-SE can extend actions with a control combination function that defaults to conjunction in the case of UML 2.

- EFFBD provides for nontriggering data inputs, which can be thought of as optional inputs (Section 2.5). A function usage can be activated without nontriggering inputs being available. In UML 2, all inputs are required to arrive, but streaming inputs may arrive after an action begins executing, and are consumed by the executing action. Optional inputs were not included in UML 2 due to concern that it can introduce inadvertent race conditions. In particular, inputs arriving late may have been intended to go with inputs that arrived earlier.

One resolution is to separate streaming as a feature from requiredness. Then they can be varied independently, providing four combinations. EFFBD supports nonstreaming inputs, both required and optional.

[6]The long-term unification of control and data functionality forms a cycle, because the new capabilities for control suggest new capabilities for data. For example, if control can have disabling values, why should data be limited being a form of enabling control? If data arrive at a function usage that is already active, it might mean that the old data were incorrect, and the function should be deactivated, and start over.

- The UML-SE requirement for discarding input values (Section 2.4) can be modeled as a characteristic of function ports (pins in UML 2), or of function inputs and outputs globally (Activity parameters in UML 2) (see Section 3.2). This can be used for traditional applications that discard enabling control values arriving at already an activated function.

Another approach to some of the issues of flexibly reacting to control and data values is to define all control and data inputs as streaming, that is, accepted by executing actions or activated function usages, and let the function determine whether inputs should be queued, discarded, required to start, and so on. This has the advantage of generality, but it would be difficult for tools to give an indication in the flow diagram of what will happen to the inputs, since this is hidden in the function.

In addition to reacting to control and data values, UML-SE requirements call for control event detection beyond traditional function completion, for example, detection of function activation, interruption, and so on. This is also generalized to events for the availability and consumption of data. UML 2 provides for events as they affect objects: change events, signal arrival, operation call arrival, and time, time being the only global event. It is not yet determined how EFFBD will model event detection, for example, whether they are specified by special flows for that purpose, or a general publish/subscribe mechanism, or rule-base representation, and so on.

3.2. Local Versus Global Specification

Since functions exist independently of the EFFBD or UML-SE flow diagrams that use them, specification elements can potentially be placed globally on functions, or locally on function usages, or both. For example, UML 2 provides for preconditions and postconditions on behavior to be placed on the behavior globally, or on individual actions invoking behavior in a flow. The advantage of local specification is that it can vary depending on where in a flow it is placed. For example, local preconditions in UML 2 activities can take into account particulars of the flow that might make additional constraints on when an action may take place, in addition to global preconditions on the behavior being invoked. The advantage of global specification is that it is more concise because it applies to all local usages of a function. For example, a global precondition on a UML 2 behavior will apply to all actions that invoke that behavior.

Three UML-SE requirements differ from UML 2 in local versus global specification:

- The specification of how input and output is handled, that is, queuing, and so on, is on function ports of the global function, whereas it is local to actions and pins in UML 2 (Section 2.4). These are the issues of Section 3.1.
- The specification of resources used by a function is defined globally (Section 2.2), whereas it is local to flows in UML 2.
- Although replication is not completely specified yet, a UML-SE requirement is that functions define the maximum number of activations that may exist at one time (Section 2.11). UML 2 currently provides for specifying whether a behavior can be executed more than once at the same time. These are both global specifications, but UML 2 is not as fine-grained as the UML-SE requirement. Additional constraints on the number of simultaneous executions in UML 2 can be locally specified, as a kind of iteration.

In either case above, UML 2 or UML-SE should provide both global and local specification elements, with global ones being shorthands for universally applied local specifications. This provides maximum flexibility for the application developer.

3.3. Other Issues

Other EFFBD and UML-SE flow diagram requirements to address in applying UML 2 are:

- A UML-SE requirement is that functions specify whether they are interruptible or not (Section 2.2).

This can be added as an additional attribute to UML 2 or UML-SE behaviors.

- A UML-SE requirement is that functions can accept inputs and provide outputs in a continuous fashion (Section 2.5). For example a function might take a position as input and provide a velocity as output and do so with values of position that vary continuously over time.

UML 2 Activities define a discrete virtual machine, but do not constrain how closely spaced in time discrete inputs may be taken and outputs provided by an action. The mathematical definition of continuity is stated the same way, in terms of limits.

However, for actions that have more than one input, a value from each input must be chosen for each execution of the action. This is not an issue for discrete

systems, since the values can be queued at each input and selected as needed for an execution. In continuous systems, the values must be selected for each execution by some other criteria, most likely the time at which the values arrive, either a global time, or time local to the action. Values arriving at the same time to each input are taken as inputs to a single execution of the action. Synchronization of this sort applies to UML joins also, which combine multiple flows into one.[7]

It will be useful to have some indication of when a flow will be continuous. This can be an additional boolean attribute on flows and notated with a keyword on the diagram, or a distinction line type.

- EFFBD and UML-SE provide for expressing probability distributions for a branch, whereas UML 2 Decision Guards do not (Section 2.8).

Flows can be extended with this information, and semantics assigned based on the distribution of data or control tokens passing through that point in the flow under some conditions.

- Multiple flows coming out of an EFFBD item node are equivalent to a UML 2 Object Node with a fork for the outgoing flows (Section 2.5).

Since this is an uncommon case, especially for physical flows, it is suggested that the fork be used when items need to be copied.

- UML 2 Object Nodes only hold objects while its containing Activity is executing, not persistently across Activity executions as in UML-SE stores or traditional data stores (Section 2.6). And objects and data flow out of UML 2 Object Nodes as they become available, rather than as they are needed. In this sense, UML 2 uses a form of data flow that treats data like a control value rather than storage.

Persistent storage can be achieved by extensions that use the predefined UML 2 actions for modifying persistent objects. Specifically, an item flow coming into a UML-SE 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 or traditional data store can be defined as an aggregate of primitive actions on persistent storage. A concise graphical notation for stores can be provided that maps to the complete action model.

[7] Streaming input and output in UML 2 is also useful in applying continuous inputs and outputs. For example, a car driver performs an action to depress the accelerator pedal, and this causes continuously varying information to be sent to the throttle, either mechanically or electronically. The action does not terminate while this output is generated, which is the definition of a streaming output in UML 2.

4. CONCLUSION

The UML 2 Activity models, EFFBD and the UML-SE requirements have similar models for control and data/item flow, and behavior in general. The UML-SE requirements specify a comprehensive set of function-based behavior, including continuous time requirements, input/output, stores, and control requirements. It brings control and data closer together in functionality than UML 2 currently does. This article identifies areas where UML 2 Activity models, EFFBD, and UML-SE requirements do not overlap and proposes changes in UML 2 or UML-SE to align them, and achieve the same execution traces for the same models. With the revisions and extensions, the functionality of EFFBD and the UML-SE requirements are fully supported by UML 2 Activities.

ACKNOWLEDGMENTS

The author thanks Sanford Friedenthal, James Long, and Joseph Skipper for input to this article.

REFERENCES

J. Axelsson, Model based systems engineering using a continuous-time extension of the Unified Modeling Language (UML), Syst Eng 5(3) (2002), 165–179.

T. Bahill and J. Daniels, Using objected-oriented and UML tools for hardware design: A case study, Syst Eng 6(1) (2003), 28–48.

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

C. Bock, UML 2 activity and action models, J Object Technol 2(4) (July–August 2003a).

C. Bock, UML without pictures, IEEE Software, special issue on model-driven development, (September/October 2003b).

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

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

F. Kockler, et al., Systems engineering management guide, 000802001202-5, Defense Systems Management College, U.S. Government Printing Office, Washington, DC, 1990.

J. Long, Relationships between common graphical representations in system engineering, ViTech Corporation, 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; December 1975. U.S. National Institute of Standards, Gaithersburg, MD.

H. Lykins, S. Friedenthal, and A. Meilich, Adapting UML for an object oriented systems engineering method (OOSEM), Proc Tenth Annu INCOSE Int Symp, Minneapolis, July 16–20, 2000.

J. Martin and J. Odell, Object-oriented analysis and design, Prentice-Hall, Englewood Cliffs, NJ, 1992.

I. Ögren, Possible tailoring of the UML for systems engineering purposes, Syst Eng 3(4) (2000), 212–224.

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

OMG (Object Management Group), UML 2.0 superstructure specification, August 2003, http://www.omg.org/cgi-bin/doc?ptc/03-08-02.

A. Pandikow and A. Törne, Support for object-orientation in AP-233, Proc Eleventh Annu INCOSE Int Symp, Melbourne, July 1–5, 2001a.

A. Pandikow and A. Törne, Integrating modern software engineering and systems engineering specification techniques, Proc Fourteenth Annu Int Conf Software & Syst Eng Appl, Paris, December 4–6, 2001.

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

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

SE-DSIG (OMG Systems Engineering Domain Special Interest Group), 2003b, http://syseng.omg.org/.

J. Skipper, private communication, 2003.

Conrad Bock is a Computer Scientist at the U.S. 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.