Model Driven Solutions

Where Business Meets Technology

A division of Data Access Technologies, Inc.
8605 Westwood Center Drive, Suite 505
Vienna, VA 22182

Prepared for

Under Purchase Order II0400381

EXECUTABLE UML/SYSML SEMANTICS

PROJECT REPORT (FINAL)

November 2008

Change Log

Version	Changes
February 2008	• Initial version. • Documented the background and overview material in Section 2.
May 2008	• Replaced the material in Section 2 with a summary of the status of the latest revised submission to the Semantics of a Foundational Subset for Executable UML Models. • Added stub overview material under Section 8.
June 2008	• Updated the status of the Semantics of a Foundational Subset for Executable UML Models submission in Section 2 for the work carried out in June.
August 2008	• Updated Section 2 to remove the submission status and, instead, reference the final submission document. • Updated Section 6 to document the discussion on the grounding of fUML semantics in logic held during the 01 August project meeting. • Proposed in Section 8 that the (few) semantic inconsistencies of fUML with the UML Superstructure specification be handled as proposed UML changes through the UML 2.3 RTF process.
November 2008	• Updated Section 2 to reference the latest Beta 1 fUML Specification. • Updated Section 3 with the specification for the semantics of streaming parameters. • Updated Section 4 with the specification for the semantics of timing. • Updated Section 5 to discuss approaches for specifying the semantics of stereotypes. • Updated Section 7 to document the fUML Reference Implementation. • Updated Section 8 to document proposed changes to UML and fUML.

Contents

Section 1 Introduction

The objective of the Executable UML/SysML Semantics Project is to specify and demonstrate the semantics required to execute activity diagrams and associated timelines per the SysML v1.0 specification and to specify the supporting semantics needed to integrate behavior with structure and realize these activities in blocks and parts represented by activity partitions. The SysML semantics will build on the Semantics of a Foundation Subset for Executable UML Models, which is currently in preparation for submission as an OMG standard.

1.1 Background

Systems engineering has a long history of using functional flow diagrams to define system behavior. An extension of functional flow diagrams called the enhanced functional flow block diagram (EFFBD) was used during the development of the SysML specification as an input to SysML activities.[1] The EFFBD is an elaboration of standard functional flow diagrams that is implemented in the Vitech CORE systems engineering tool[2] and can be executed using their simulation capability. Their capability also enables generation of a detailed timeline of the execution.

Many UML based tools do have execution capability with sequence diagrams and state machines. However, the current SysML tools still are inadequate in their ability to provide executable activity diagrams and associated timelines, even though this was considered an important requirement in the behavior requirements in the UML for Systems Engineering RFP[3] Part of the problem is due to limitations of the specification, and part of the problem is due to tool vendor limitations. This limitation, in turn, limits the ability to use SysML to analyze, specify, verify, and validate system requirements and design.

Establishing clear and concise execution semantics for an activity diagram is a first step. This includes the ability to specify the execution for both control flow and data flow. In particular, an important capability is to be able to specify the execution semantics for both streaming and non-streaming inputs and outputs, as well as required and optional inputs and outputs. There are associated issues that also need to be addressed to begin to realize the full potential of SysML, such as how the behavior of activities using swim lanes (e.g. activity partitions) can be realized in the system design model. This includes the implications of executable activity semantics on ports semantic (primarily SysML flow ports, but also standard ports). There are other related issues to be addressed to enable the integration of behavior and structure.

It is expected that specifying these precise execution semantics for SysML can build on the work done in on the submission in response to the Object Management Group (OMG) request for proposal (RFP) on “Semantics of a Foundational Subset for Executable UML Models”. It is understood that a substantial amount of the work has been done, but some additional work is

[1] Refer to the "SysML and UML 2.0 Support for Activity Modeling" (http://www.mel.nist.gov/msidlibrary/doc/sysmlactivity.pdf) by Conrad Bock on the OMG SysML website at http://www.omgsysml.org.
[2] See http://www.vitechcorp.com/.
[3] See http://www.omg.org/cgi-bin/doc?ad/2003-3-41.

needed to complete the specification. In addition, further work is required beyond the initial scope of the executable UML to fully address the ability to execute SysML activity diagrams and integrate them with the structural semantics per above. This includes addressing other aspects of execution, such as the streaming I/O and timing diagram, port semantics to integrate with structure, and other aspects considered essential for systems engineering.

1.2 Report Organization

This report contains a section covering each of the required tasks of the project.

1. Foundational UML execution semantics
2. SysML activity execution
3. Activity timing diagram
4. Structured model execution
5. Formal operational semantics
6. Reference Implementation

Per the project SOW, this document will be updated at the end of each month and delivered as part of a status report showing progress made during the month. The final version at the end of the project will service as the final report for the project.

1.3 References

The following specifications are referenced throughout the remainder of this document.

- *OMG Unified Modeling Language (OMG UML), Superstructure,* Version 2.2, February 2009 (formal/09-02-02) [referenced in the following as the *UML 2 Superstructure Specification*]
- *OMG Systems Modeling Language (OMG SysML),* Version 1.0, September 2007 (formal/07-09-01) [referenced in the following as the *SysML Specification*]
- *Semantics of a Foundational Subset for Executable UML Models*, FTF – Beta 1, November 2008 (ptc/09-11-03) [referenced in the following as the *fUML Specification*]

Section 2 Foundational UML Execution Semantics

According to the project SOW, the Foundational Executable UML Semantics task is required to:

> *"Complete the specification of the semantics of the Foundational UML subset as required to submit a response to the OMG RFP for the Semantics of a Foundational Subset for Executable UML Models, and as the foundation for the further semantics of activity modeling required for SysML. The Foundational UML subset includes call behavior actions, call operation actions, send signal actions, accept event actions and a library of basic arithmetic functions."*

A second revised response to the OMG RFP for the *Semantics of a Foundational Subset for Executable UML Models* was submitted to OMG on 25 August, 2008 and was subsequently adopted by the OMG Board of Directors. A Finalization Task Force (FTF) has been chartered for the specification. The currently available version of the specification is the FTF Beta 1 version, a copy of which is attached to this report.

Section 3 SysML Activity Execution

According to the project SOW, the SysML Activity Semantics task is required to:

> *"Extend the Foundation UML specification of activity model semantics to include streaming, optional I/O and other UML 2.0 and SysML activity modeling capabilities, per the UML 2 and SysML v1.0 specification. The semantics should be specified in a clear and concise way that can be implemented by a UML/SysML tool vendor. This should include the behavioral semantics (vs. structural semantics) that defines the instances and how they change over time. The specification should be in a format that is consistent with an OMG submission in response to an Executable UML/SysML RFP."*

The semantics to be specified in this task are for certain constructs that are included in the UML4SysML subset defined in the *SysML Specification,* Subclause 4.2. The semantics for these constructs are defined as an extension to the execution model defined in the *fUML Specification,* Clause 8. Following a similar organizational approach to the fUML model, the UML4SysML semantic models are grouped under a UML4SysML::Semantics package, with a sub-package structure that parallels that of the fUML::Semantics package.

Of the areas outlined in the SOW for this task, the only semantic specification actually completed during the period of performance of the project was for streaming parameters. The specification for these semantics is given below.

3.1 Streaming Parameters

3.1.1 Abstract Syntax

There are no structural differences between the UML4SysML abstract syntax and the fUML abstract syntax related to streaming. Instead, there are just two constraints in the fUML subset that prevent streaming parameters, but which are not present in UML4SysML.

- fUML requires that the isStream attribute of a parameter be false (see the *fUML Specification,* Subclause 7.2.2.2.26, Additional Constraint [1]). For a streaming parameter, this must be set to true.
- fUML requires that the isReentrant attribute of a behavior be true (see the *fUML Specification,* Subclause 7.3.2.2.1, Additional Constraint [1]). A behavior with streaming parameters is required to have isReentrant be true (see the *UML 2 Superstructure Specification,* Subclause 12.3.41, Constraint [3]).

3.1.2 Semantics

Updates to the fUML Execution Model required to specify the semantics of streaming parameters are contained in the UML4SysML semantics packages CommonBehaviors::BasicBehaviors, Activities::IntermediateActivities and Actions::BasicActions. Section 3.1.2.1 below gives an overview to the approach for this specification, followed by a description of the model included in each semantic package.

3.1.2.1 Overview

Background

As far as the execution of activities is concerned, it is the semantics of call actions that are primarily affected by streaming parameters. First, consider calling a behavior without streaming parameters (either directly by a call behavior action or as an operation method through a call operation action). In this case values on the input pins of the calling action are made available to the executing behavior, the behavior executes to completion and any output parameter values are placed on the output pins of the calling action.

For example, consider the simple activity shown in Figure 3-1 and the call to this activity shown in Figure 3-2. Figure 3-3 shows a portion for the semantic representation of the call behavior action shown in Figure 3-2. As indicated in the diagram, the model includes two *activity node activation groups* (for more on activation groups, see the *fUML Specification,* Subclause 8.5.2.1). One group contains the activations for activity nodes in the activity enclosing the call behavior action. The other group contains the activations for activity nodes in the Example activity being called.

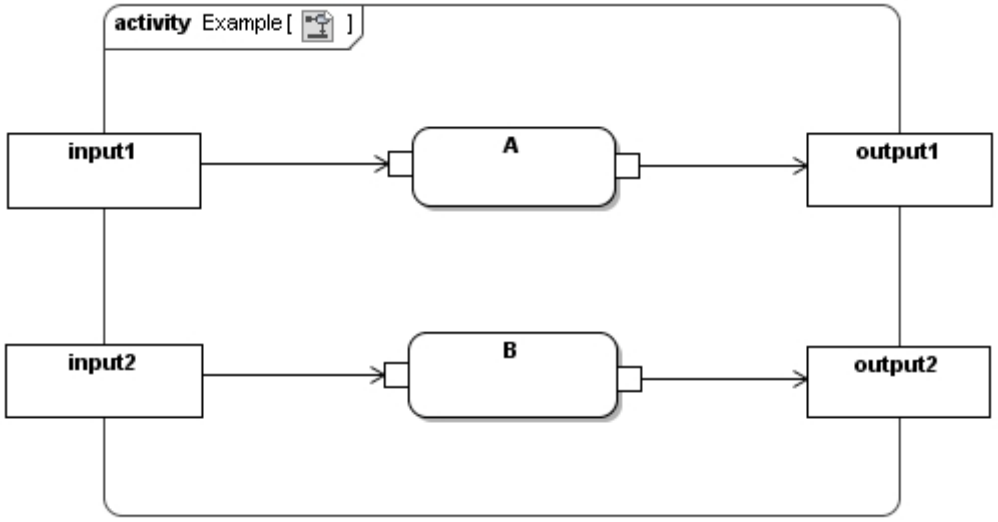

Figure 3-1 A Simple Activity

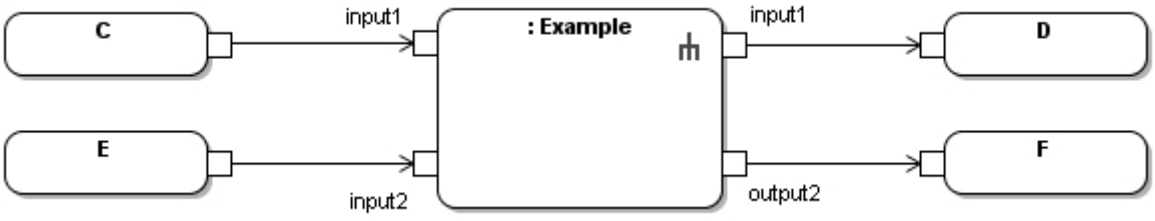

Figure 3-2 A Call of the Example Activity

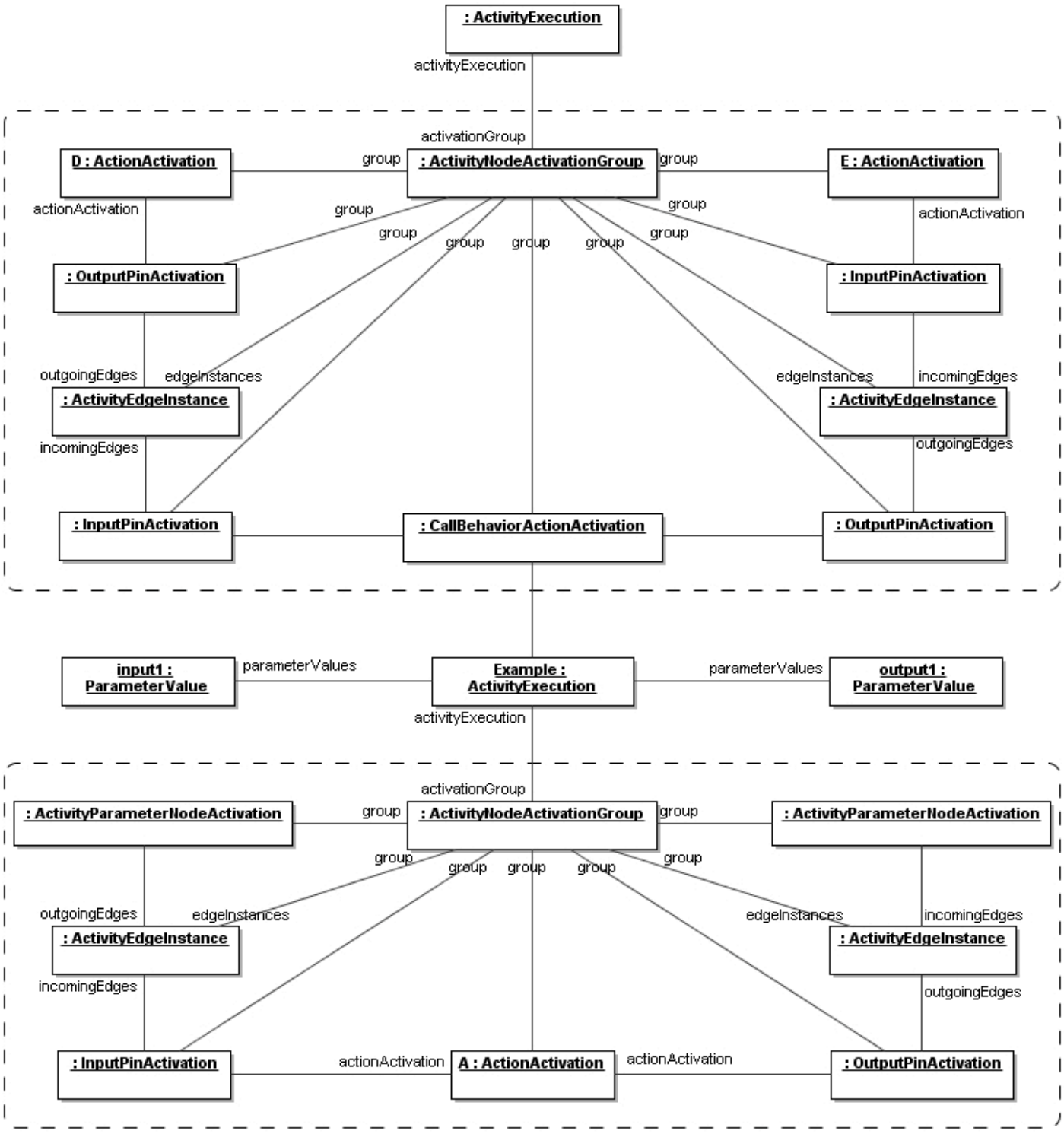

Figure 3-3 Semantic Model for Call Behavior Action Activation

The model in Figure 3-3 is per the *fUML Specification,* which does not provide semantics for streaming. Note that there are no direct connections between the call action pin activations in the activation group for the calling activity and the corresponding activity parameter node

activations in the activation group for the called activity. Instead, when the call behavior action fires, it takes values from its argument input pin activations and places them in the parameter value objects for the corresponding parameters, from where they are available to the activity parameter nodes for those parameters. When the Example activity has completed execution, the ActivityExecution::execute operation takes the values from the activity parameter nodes for output parameters and places them in the corresponding parameter value objects, from which the call behavior action can obtain them to place on its result output pin activations. Thus, as mentioned previously, input parameter values flow into the called activity only at the start of execution, and output parameter values flow out only at the end.

On the other hand, if a behavior with input streaming parameters is called, then the behavior may continue executing even after all non-streaming inputs have been processed, in order to handle additional inputs that may be received through the streaming parameters. Further, if the behavior has streaming output parameters, then values posted to those parameters will be immediately offered from the output pins of the calling action, even while the behavior is still executing. The fUML semantic model for call actions needs to be extended to allow these semantics.

Semantic Analog for Streaming

There is a close similarity between the semantics of calling an activity the semantics of a structured activity node with a similar body to the behavior. To a certain extent, the semantics of such a call can be thought of as being much like doing a "macro insertion" of the body of the called activity as a structured activity node at the point of the call. We can use this similarity to develop an analog for extending call action semantics to handle streaming parameters, too.

For example, suppose the input1 and output1 parameters of the Example activity in Figure 3-1 are now considered to be streaming. Figure 3-4 shows the semantically similar structured activity node expansion corresponding to the call action in Figure 3-2, *with* the appropriate semantics for the streaming parameters.

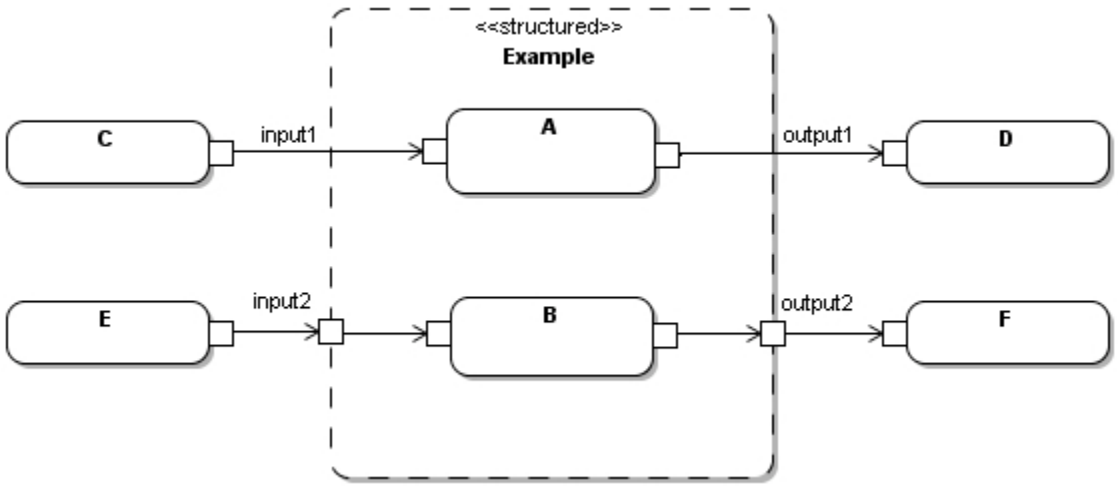

Figure 3-4 A Semantically Similar Structured Activity Node

Note that the pins for non-streaming parameters on the call action have corresponding pins on the structured activity node. This means that the normal rules for firing and completing the structured activity node as an action apply to these pins. On the other hand, the pins for

streaming parameters on the call action do *not* have corresponding pins on the structured activity node. Instead, the incoming object flow from action C passes directly into the interior of the structured activity node and the outgoing object flow to action D passes directly out. This reflects the semantics that, once the structured activity node has fired, tokens can pass freely in and out on these flows.

The fUML specification already handles the semantics of structured activity nodes. Figure 3-5 shows a part of the representation in the fUML Execution Model of the execution of the model from Figure 3-4. As shown, the execution again includes two activity node activation groups. This time, the first group contains the activity node activations for activity nodes directly within the enclosing activity for the model fragment shown in Figure 3-4. The second group contains the activity node activations for the activity nodes nested within the structured activity node.

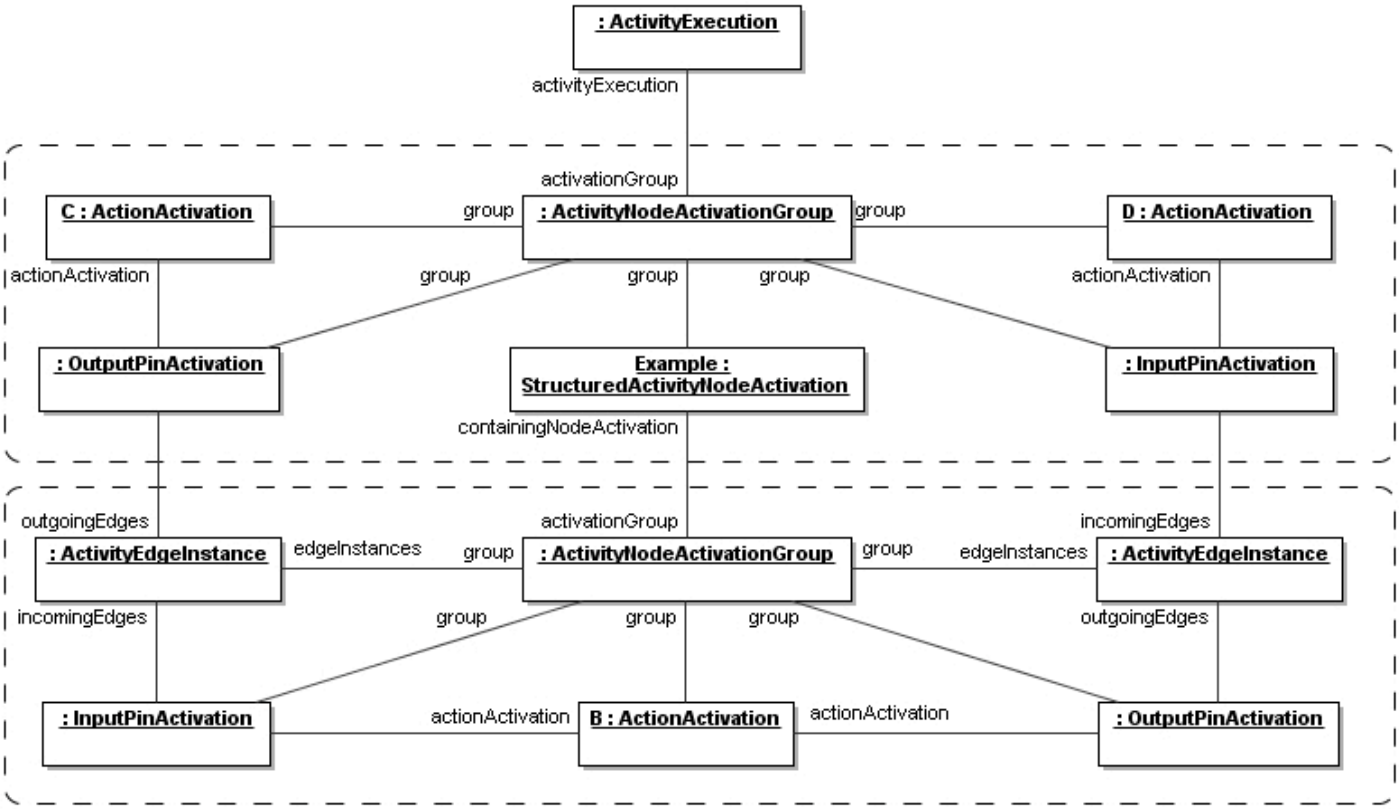

Figure 3-5 Semantic Model for Structured Activity Node Activation

The activity node activation group for the structured activity node activation is created when the structured activity node activation fires. At this time, activity edge instances are also created for the object flows that cross the structured activity node boundary in Figure 3-4. As shown in Figure 3-5, these edge instances directly connected pin activations outside the nested activity node activation group to those inside, allowing the flow of tokens regardless of the presence of the surrounding structured activity node.

Semantic Approach for Streaming

The semantics of the call action with streaming parameters is analogous. Once the call action has fired, tokens can continue to pass in and out on the flows attached to pins corresponding to streaming parameters, during the execution of the called activity. In order to allow such

streaming semantics, it is necessary to create connections across the two activity node activation groups shown for the call action activation in Figure 3-3 in order to allow values to stream in and out of the called activity during its execution.

The necessary connections are provided by introducing the concept of *listeners* for streaming parameter values (see Figure 3-8). A streaming parameter value has a set of registered listeners. When values are posted to the streaming parameter, they are distributed to each of the registered listeners. For the purposes of activities, a specialized streaming parameter listener is used that fires a specific activity node activation with tokens containing the posted values it receives (see Figure 3-9).

In the case of an input parameter, the activity parameter node creates such a listener for itself and registers it with the appropriate parameter value. The call action input pin activation then continues to post values to the streaming parameter value as it receives them at any time during the execution of the call action, and these are then forwarded via the listener to the activity parameter node to be offered within the called activity.

For an output parameter, the call behavior action creates a listener for its corresponding output pin activation. The activity parameter node activation for the output parameter within the called activity execution then posts values to the streaming parameter value as it receives them at any time during the activity execution, and these are then forwarded via the listener to the output pin activation to be offered within the calling activity.

Figure 3-6 shows how the listeners are used to make the connection between the two activity node activation groups in Figure 3-3. Note that there is no direct connection between the activity parameter node activation for an output parameter and the streaming parameter value object for that parameter. Instead, the activity parameter node activation obtains the appropriate parameter value from its activity execution when necessary to post new values.

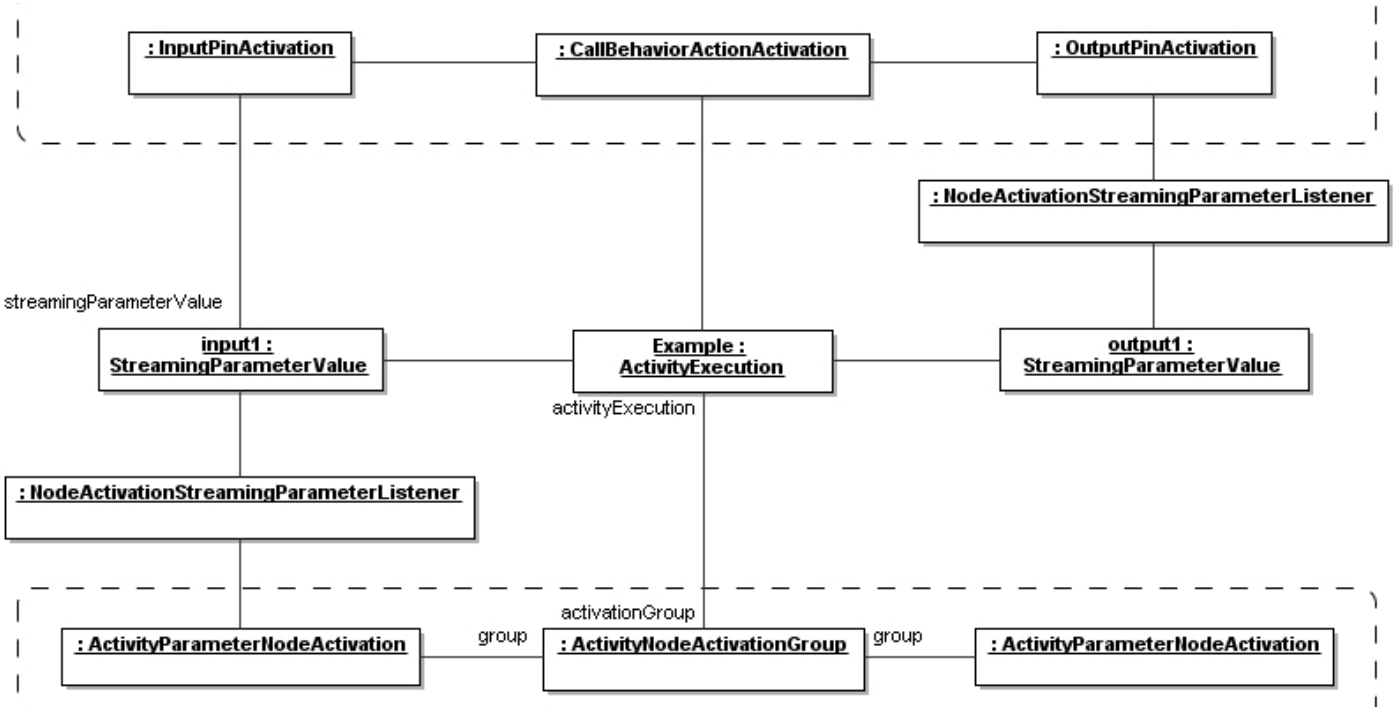

Figure 3-6 Semantic Model for Streaming using Listeners

Reentrancy

The *UML 2 Superstructure Specification* requires that behaviors with streaming parameters be non-reentrant (see constraint [3] under Subclause 12.3.41). The description for the Behavior::isReentrant attribute is "Tells whether the behavior can be invoked while it is still executing from a previous invocation" (Subclause 13.3.2). The reason for the constraint is given as "A reentrant behavior cannot have streaming parameters because there are potentially multiple executions of the behavior going at the same time, and it is ambiguous which execution should receive streaming tokens" (Subclause 12.3.41).

However, the description of reentrancy for behaviors would seem to imply that the behavior cannot be invoked again if *any* invocation of it is currently executing. This would mean that if one call action for the behavior invoked it, then not only that call action but all *other* call actions will be blocked from invoking the behavior until the first invocation has completed.

But stringent a restriction is not necessary for the purposes of streaming. For a specific call action, it is completely unambiguous which behavior execution should receive streaming tokens (i.e., the one associated with that call action activation), even if the same behavior is invoked from other call actions. The only restriction required is that the behavior not be allowed to be invoked from the *same* call action activation again while a previous invocation is executing. That is, a call action to a behavior with streaming parameters is blocked from firing again while a previous invocation of the behavior is still executing.

The semantics for streaming presented here presumes this more limited restriction. It will be proposed to the UML 2.3 Revision Task Force that the semantics for reentrancy in UML 2 be revised to reflect this (see Section 8).

Limitations

The fUML semantic model for call action activation includes a common CallActionActivation superclass, with CallBehaviorActionActivation and CallOperationActionActivation. The CallActionActivation::getCallExecution operation is abstract, with a method for it provided by each concrete subclass. This operation returns an execution object for the behavior that is being called by the call action. For a call behavior action, this is the specific behavior given in the action. For a call operation action, this is the method resulting from polymorphic dispatching of the operation given in the action.

The current semantic model for streaming maintains the getCallExecution subclass interface for CallActionActivation. However, this operation cannot be called in the superclass until the action actually fires, since it is only then that the called behavior actually executes. This means that the superclass does not have access to the execution, or even its associated behavior, until this time (since, in particular, which behavior is executing for an operation may depend on the target object of the call).

Since the CallActionActivation superclass does not have access to the behavior being executed, the fact that one or more parameters of that behavior are streaming cannot be used in the determination when to fire. This means that the normal rules on when to fire will be used for call action input pins for streaming parameters, rather than the special rules given in the *UML 2 Superstructure Specification,* Subclause 12.3.41. In particular, a call action will not fire until the number of values are offered to an input pin is at least equal to its multiplicity lower bound, even if the pin is for a streaming parameter.

Further, while this is not stated explicitly in the superstructure specification, it would seem that values beyond the multiplicity upper bound for an input pin for a streaming parameter that arrive during the course of a call action activation should not be posted during that activation. Further, no check is currently made on the whether the number of values posted to streaming output parameters conforms to the multiplicity of those parameters.

For these reasons, the current specification for streaming parameter semantics is limited to the case in which all streaming parameters have multiplicity 0..*. This has the implication that, once a call action activation for a call to a behavior with streaming input parameters has fired, the called behavior will continue to execute, accepting streaming inputs, until the enclosing activity or structured activity node of the call action terminates. Such a call action activation will thus never fire more than once.

This limitation can be removed in the future by adding an operation to the subclass interface for CallActionActivation to allow the superclass to obtain the list of parameters for the behavior or operation to be called, before the action actually fires. The CallActionActivation class can then override the isReady operation inherited from ActionActivation to specify the specialized logic required to properly handle streaming parameters.

3.1.2.2 Loci

The specification of the semantics for streaming requires specializations or replacements for the semantic visitor classes for four concrete syntactic elements: ActivityParameterNode, InputPin, CallBehaviorAction and CallOperationAction. As shown in Figure 3-7, a specialized ExecutionFactory class is used in order that the specialized UML4SysML semantic visitors are instantiated for these elements. The specialized ExecutionFactory class overrides the inherited *instatiateVistor* operation, checking specifically if the given element is one of the four listed above, in which case the appropriate UML4SysML semantic visitor class (as specified in the following subsections) is instantiated and returned. For all other syntactic elements, the fUML ExecutionFactory::instantiateVisitor operation is used to generate the standard fUML semantic visitor.

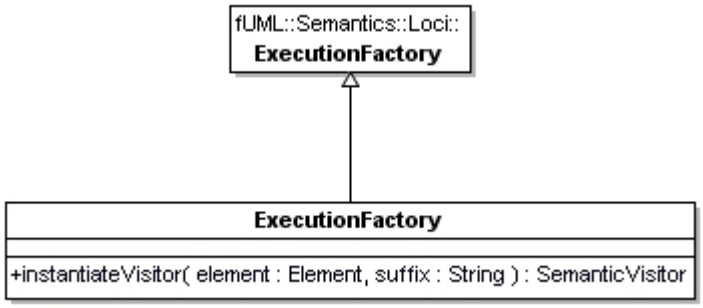

Figure 3-7 ExecutionFactory for UML4SysML

3.1.2.3 Common Behaviors

The UML4SysML CommonBehaviors::BasicBehaviors package contains the StreamingParameterValue subclass of the fUML ParameterValue class (see Figure 3-8). This subclass adds the ability to register *listeners* for a parameter value. After the parameter value is

initially established, new values may be *posted* to it during the course of the associated execution. These new values are then forwarded to all the listeners.

Listeners for streaming parameter values have the class StreamingParameterListener as their type. This is an abstract class, in order to decouple the Basic Behaviors model from other parts of the Execution Model which may wish to register listeners. Thus, concrete subclass of StreamingParameterListener are modeled specifically in those other areas (particularly NodeActivationStreamingParameterListener for activities—see Section 3.1.2.4).

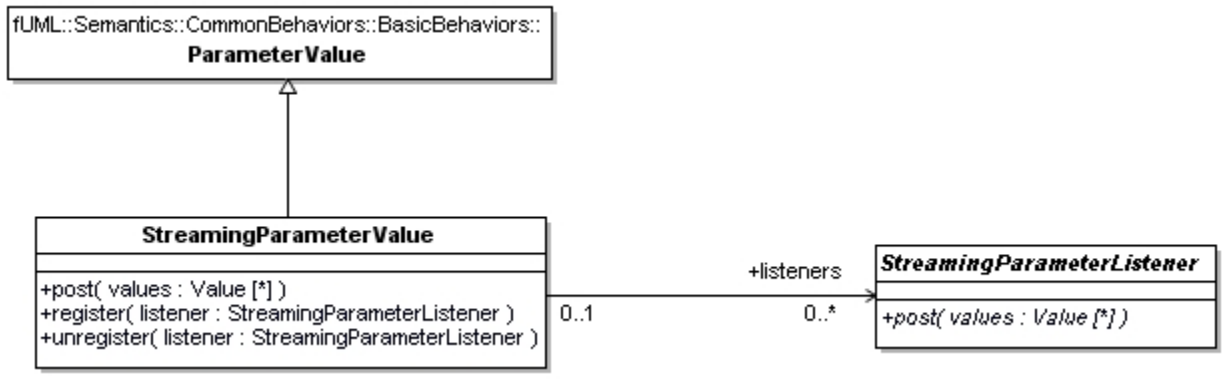

Figure 3-8 Streaming Parameter Values (BasicBehaviors)

3.1.2.4 Activities

The UML4SysML Activities::IntermediateActivities package contains a specialization of the fUML ActivityParameterNodeActivation class and a concrete NodeActivationStreamingParameterListener subclass of StreamingParameterListener (see Figure 3-9). The specialized ActivityParameterNodeActivation class overrides the inherited *run* and *fire* operations.

- *run.* If the activity parameter node is for a streaming input parameter, then the specialized run operation creates a node activation streaming listener for the activity parameter node activation and registers it with the appropriate streaming parameter value. In this way, values posted to the input parameter will be forwarded via the listener, resulting in a firing of the activity parameter node activation.
- *fire.* If the activity parameter node is for a streaming output parameter, then the specialized fire operations posts values from the tokens it receives to the appropriate streaming parameter value. In all other cases, the functionality is the same as the fUML ActivityParameterNodeActivation fire operation.

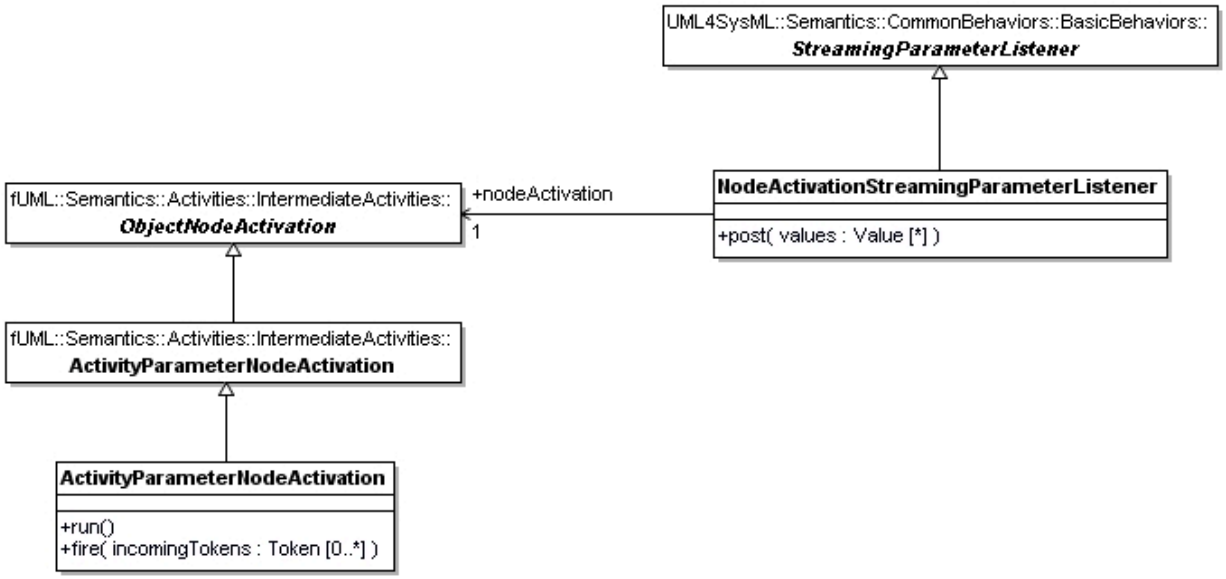

Figure 3-9 Activity Parameter Node Activations with Streaming (IntermediateActivities)

3.1.2.5 Actions

The UML4SysML Actions::BasicActions package contains a specialization of the fUML InputPinActivation class and a replacement for the fUML CallActionActivation class (see Figure 3-10). In addition, it contains replacements for the fUML CallBehaviorActivation and CallOperationActivation classes that specialize the UML4SysML CallActionActivation superclass, rather than the original fUML superclass. However, other than this, the subclass functionality of the CallBehaviorActivation and CallOperationActivation classes is identical to that of the corresponding fUML classes.

The specialized InputPinActivation class adds a *streamingParameterValue* attribute. If the input pin activation is for an input pin of a call action corresponding to a streaming parameter, and the call action has fired, then the streamingParameterValue attribute will contain a reference to the appropriate streaming parameter value of the execution of the called behavior. The specialized class also overrides the inherited *receiveOffer* operation so that, if there is a streaming parameter value reference, any offered tokens are immediately accepted and their values posted to the streaming parameter value. If there is no streaming parameter value reference, then the fUML InputPinActivation receiveOffer operation is used (which simply forwards the offer to the action activation for the input pin activation).

The specialized CallActionActivation overrides the *isReady, doAction* and *terminate* operations it inherits from ActionActivation.

- *isReady.* The call action activation is not considered ready if there it is currently executing a non-reentrant behavior.
- *doAction.* If the called behavior has streaming parameters, then streaming parameter values are created for them and set in the execution object. Further, for input pin activations corresponding to streaming parameters, the reference is set for the pin activation to the appropriate streaming parameter value. Finally, after the initial thread of execution, output parameter values are offered on the output pins of the call action only if the called behavior

does not have streaming input parameters. If the called behavior does have streaming input parameters, then its execution remains in place, in order to process any additional inputs that may be posted to those parameters.

- *terminate.* On forced termination of the call action activation, any pending executions for called behaviors are both terminated and destroyed. (The standard fUML CallActionActivation::terminate operation only terminates, but does not destroy such execution. This is because the execution is always destroyed within the CallActionActivation::doAction operation. However, for an execution of a behavior with streaming input parameters, this operation may have already returned, so it is necessary to insure that all called execution objects are always destroyed.)

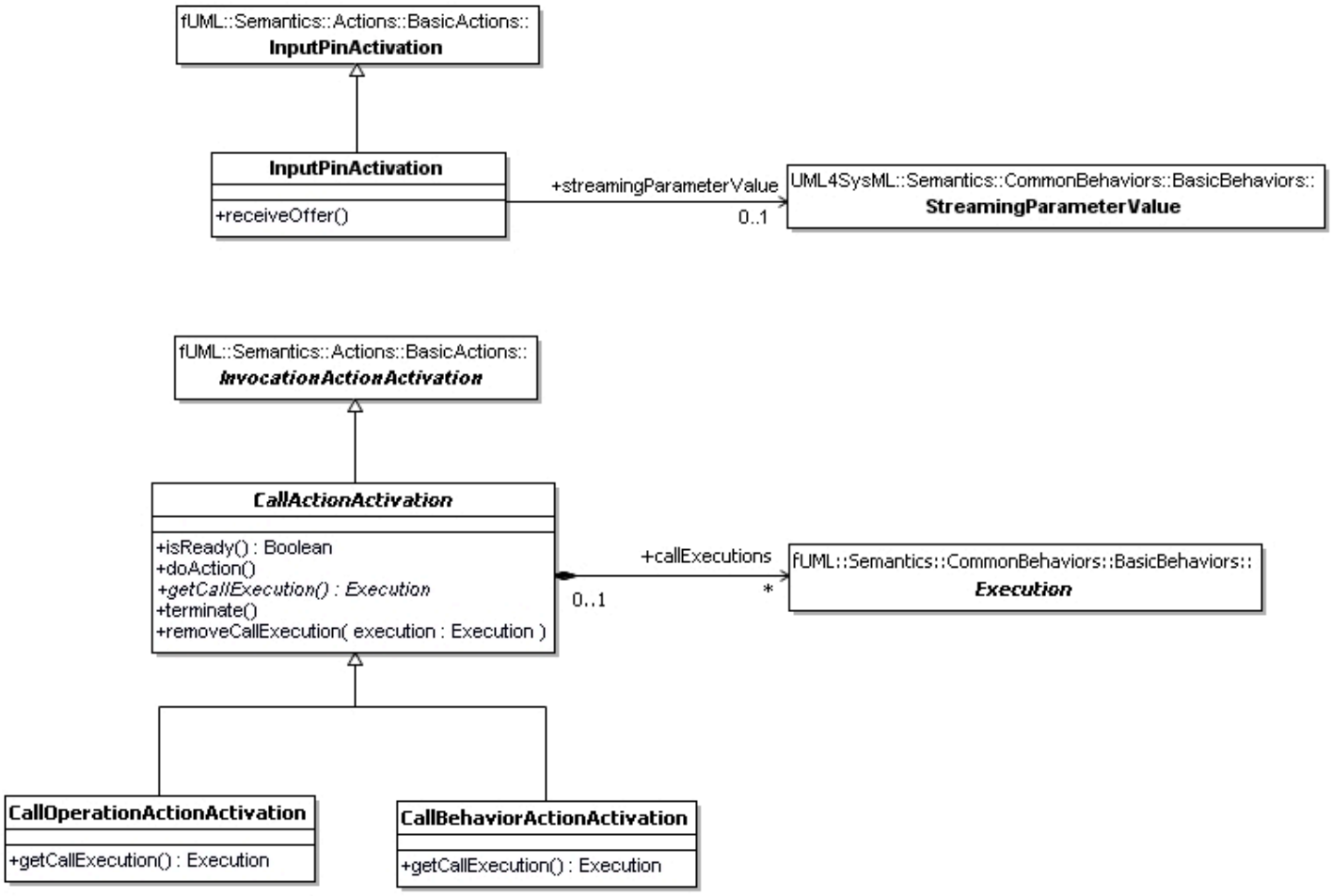

Figure 3-10 Call Action Activation with Streaming (BasicActions)

NOTE: As shown in Figure 3-10, the UML4SysML CallActionActivation::callExecutions attribute has a multiplicity of *, even though the corresponding attribute on the fUML CallActionActivation class only has a multiplicity of 0..1 (see the *fUML Specification,* Subclause 8.6.2). The multiplicity of * allows a single call action activation to fire multiple times resulting in multiple concurrent invocations of a reentrant behavior. The multiplicity of 0..1 in the fUML specification is actually an error (since all behaviors are required to be reentrant in fUML). This will be proposed to be corrected by the fUML Finalization Task Force (see Section 8).

3.1.2.6 Class Descriptions

See the UML4SysML semantics model file for the descriptions of the above classes and Java code specifications for all operations.

Section 4 Activity Timing Diagram

According to the project SOW, the Activity Timing Diagram task is required to:

> *"Specify the semantics for the generation of a timeline for the executable activity. The timeline should leverage the UML/SysML simple time model. The assumptions for concurrency should enable a best case (parallel) and worst case (sequential) timeline. (Note: It is assumed that future work is needed to define the semantics for a case with an assumed probability distribution on the time durations and/or start and stop times)."*

Subclause 2.3 of the *fUML Specification* lists the following as one of the items "not constrained by the execution model":

- *The semantics of time*. The execution model is agnostic about the semantics of time. This allows for a wide variety of time models to be supported, including discrete time (such as synchronous time models) and continuous (dense) time. Furthermore, it does not make any assumptions about the sources of time information and the related mechanisms, allowing both centralized and distributed time models.

Effectively, however, the fUML Execution Model is written (in the *fUML Specification,* Clause 8) as if execution happens instantaneously (when not waiting to be triggered by some event). To specify the semantics for the generation of a timeline for activity execution, it is necessary to extend the execution model to explicitly account for the semantics of time. Further, in order to be able to specify timing constraints in a user model, it is necessary to extend the fUML abstract syntax to include such constraints.

The semantic model for timing is organized as a further extension to the UML4SysML semantics described in Section 3, rooted in a UML4SysMLWithTime package. Again following a similar organization to the fUML model, the UML4SysMLWithTime semantic models are grouped under a UML4SysMLWithTime::Semantics package, with a sub-package structure that parallels that of the fUML::Semantics package.

Due to a difficulty with the current implicit threading approach in the fUML Execution Model (as described further in Section 4.3.3), a fully operational specification for executing actions with timing constraints was not completed within the period of performance of the project. Nevertheless, the overall approach described in this section provides a largely complete basis for such a specification.

4.1 Abstract Syntax

To produce a timeline for the execution of an activity, it is necessary to know how long nodes in the activity take to execute. The most direct way to do this is to attach a duration constraint to an executable node within an activity in order to specify a non-zero execution duration for it. Therefore, the fUML subset needs to be extended to include at least a basic form of such constraints. Further to operationally allow for delays in time, the fUML constraint on accept event actions needs to be relaxed to allow for triggering by (relative) time events, in addition to just signal events.

4.1.1 Classes

The fUML subset does not include constraints (see *fUML Specification,* Subclause 7.2.2.1). The UML4SysML subset includes Constraint, but it is only needed here as an ancestor class of DurationConstraint (see Section 4.1.2 below), and cannot be used otherwise. As shown in Figure 4-1, note the following differences from the full Constraint class in UML2.

- The Namespace::ownedRule association is not included. This is to avoid modifying the Namespace class from the fUML abstract syntax subset (since association ends are always owned by the classes in the abstract syntax model).
- The Constraint::specification association is not included. This is redefined in IntervalConstraint and then again in DurationConstraint. Since redefinition is not used in the fUML abstract syntax model (see *fUML Specification,* Subclause 8.1, for the fUML conventions on handling redefinition), the specification property is not included for the superclasses and is only defined for the ultimately desired subclass, DurationConstraint (see Section 4.1.2).

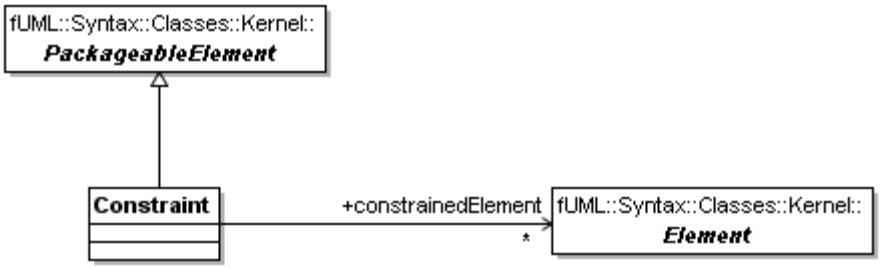

Figure 4-1 Constraints

4.1.2 Common Behaviors

In SysML, the UML4SysML (see *SysML Specification,* Subclause 4.2) package merges the entire UML 2 SimpleTime package (see *UML 2 Superstructure Specification,* Clause 13, particularly Figure 13.13). However, the semantics provided in this specification only covers a limited subset of the SimpleTime syntax. Specifically, duration constraints are supported on actions and time expressions are supported on time events used to trigger accept event actions.

Figure 4-2 shows the classes from the SimpleTime package that are covered here, modeled as extensions to the fUML abstract syntax subset. The usage of these classes, along with additional constraints required to be meet, are listed below.

- *Duration.* The time model assumed here is limited to discrete time (see Section 4.2.1 below). A duration can therefore be given as a non-negative integer. Only the specification of constant durations is supported.
    - A Duration must have an expr that is an IntegerLiteral with a non-negative value.
    - A Duration must have no observations.
- *DurationInterval.* A duration interval defines a Duration range, as used in a DurationConstraint.

- *DurationConstraint.* A duration constraint requires that the duration of execution of the constrained element be within the range given by a duration interval. Only duration constraints on actions are supported.
    - A DurationConstraint must have a single constrainedElement that is an Action, but not an AcceptEventAction.
    - The DurationConstraint must have as its namespace the Activity that owns the Action that is its constrainedElement.
- *Interval.* Interval is included as the superclass of DurationInterval,
- *IntervalConstraint.* IntervalConstraint is included as the superclass of DurationConstraint.
- *TimeEvent.* A time event specifies a point in time. Only relative time events are supported, in which the point in time is given by an elapse time relative to some other (implicit) start time.
    - A TimeEvent must have isRelative equal true.
- *TimeExpression.* TimeExpression is included solely for use as part of a TimeEvent, in which the expression must be a non-negative constant giving the relative elapse time for the event. (Note that TimeDurations and TimeConstraints are not supported.)
    - A TimeExpression must have an expr that is an IntegerLiteral with a non-negative value.
    - A TimeExpression must have no observations.

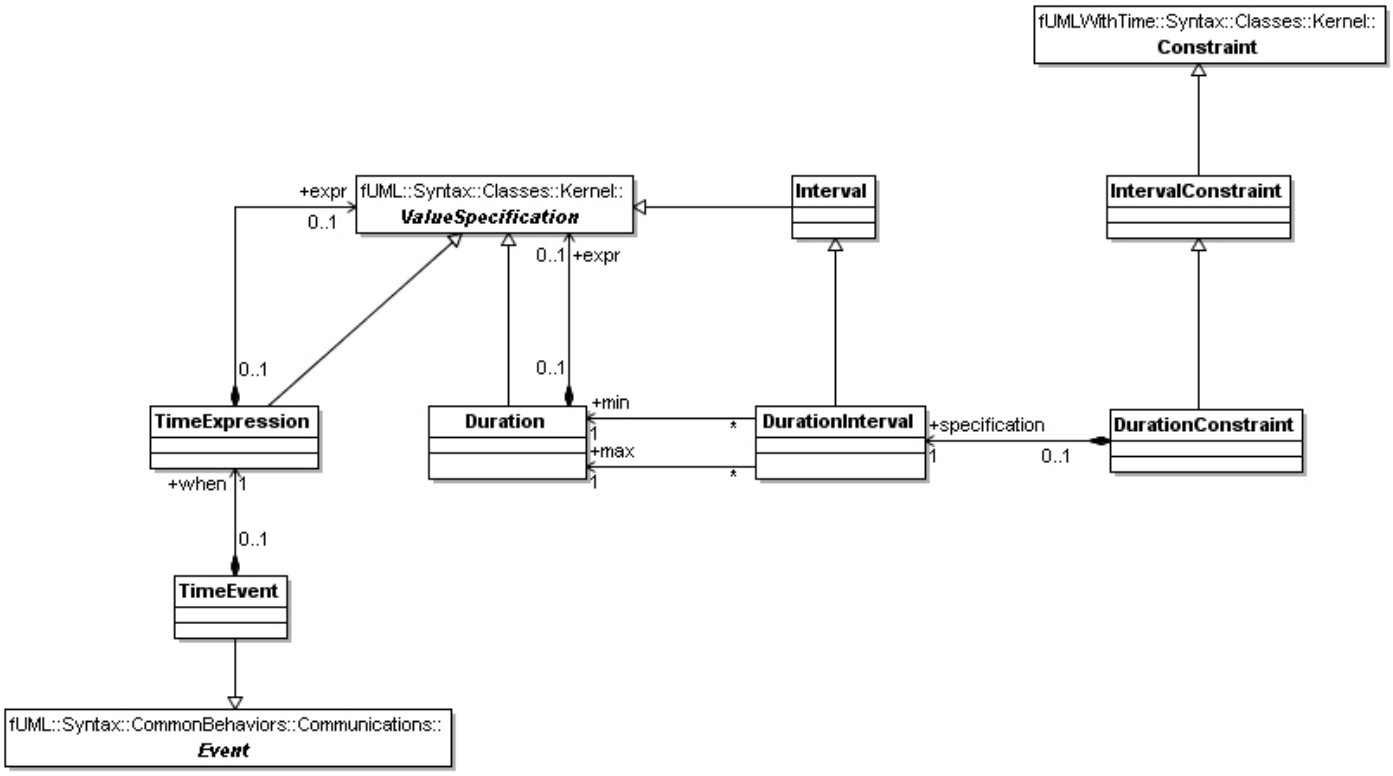

Figure 4-2 SimpleTime

4.1.3 Actions

fUML requires that accept event actions have triggers for only signal events (see the *fUML Specification,* Subclause 7.5.4.2.1, Additional Constraint [2]). This needs to be relaxed to allow an accept event action to be triggered by a time event:

- An AcceptEventAction must either have triggers that are all for signal events or a single trigger for a time event.

4.2 Model Library

While the SimpleTime package defines the concept of a time expression, the UML superstructure does not provide any built in Time type. In order to formalize the concept of time, it is necessary to provide such a type and a library of functions to operate on it.

4.2.1 Time Functions

The TimeFunctions package defines the primitive type Time and provides the function behaviors listed in Table 4-1, which act as primitive functions operating on this type. These primitive functions should be registered with the execution factory when initializing a fUML execution environment (see *fUML Specification,* Subclause 8.2.1), so they are available for invocation by call behavior actions. Note that a primitive function for equality is not required, since the semantics of time values (see Section 4.3.2), as instances of a primitive type, allow their equality to be tested using a test identity action.

Table 4-1 Time Functions

Function Signature	Description
`EarliestTime( ): Time`	The earliest representable time.
`CurrentTime( ): Time`	The current time (see also Section 4.3.1 below).
`+(time: Time, interval: Integer): Time`	The time later than the given time by the given interval. Note that the interval may be negative, in which case the returned time is earlier than the given time, but not earlier than the earliest time.
`-(time1:Time, time2: Time): Integer`	The interval between the two given times. This is the interval which, if added to the first time, gives the second time.
`<(time1: Time, time2: Time): Boolean`	True if time1 is earlier than time2.
`<=(time1: Time, time2: Time): Boolean`	True if time2 is not later than time1.
`>(time1: Time, time2: Time): Boolean`	True if time1 is later than time2.
`>=(time1: Time, time2: Time): Boolean`	True if time2 is not earlier than time1.

4.3 Semantics

The execution semantics for timing must specify two things. First, it must operationally define the concept of the *flow of time.* Second, it must ensure that an action with a duration constraint actually have an execution duration that is within that required by the constraint. The first if these things is handled by specifying the semantics of *clocks* and *time values* (see Sections 4.3.1 and 0), while the second is dealt with by extending the semantics of the execution of actions (see Section 4.3.3).

4.3.1 Loci

The UML4SysMLWithTime Loci package adds a model of *clocks* and *threads* to be associated with an execution locus.

Clocks

In the context of the fUML execution model, a *clock* is an active object with an operation that may be called to obtain the *current time* (see Figure 4-3). A clock is started at an initial time using the *start* operation, after which the clock continually updates the time using the behavior shown in Figure 4-4. This behavior simply waits for a duration of 1, advances the current time and then loops back to wait again.

Note that the behavior in Figure 4-4 uses an accept event action triggered by a time event in order to specify a wait duration. Since this activity is part of the execution model, the semantics for this accept event action needs to be interpreted in terms of the base semantic formalism (*fUML Specification,* Clause 9). Thus, the base semantics need to be extended to allow accept event actions triggered by (relative) time events (see Section 6.2). This is the only extension to the base semantics required in order to introduce the concept of the flow of time into the execution model.

As also shown in Figure 4-3, the Locus class is specialized to require each locus to have a single clock. That is, each locus effectively defines its own “flow of time”. The primitive *CurrentTime* function (see Section 4.2.1) is defined to return the current time of the clock for the locus at which the function is invoked.

However, since all clocks are ultimately based on the common underlying time model of the base semantics, it would seem that clocks at different loci would still be, *a priori,* synchronized. In order to allow for the possibility of *unsynchronized* clocks at different loci, the clock model includes the concept of an *indeterminate time step.* This means that a single clock step may actually have a duration longer than 1 in the underlying base semantic time model, but that the exact duration is indeterminate. In this way, even though each clock seems to advance smoothly at its own locus, the time readings of one clock do not have any determinate relationship to the readings of another clock—other than that all clocks still advance monotonically relative to the underlying flow of time in the base semantics.

To model this operationally, each clock is given a *maximum step duration* when it is started. The *wait duration* for the clock is then set to a value from 1 to this maximum, using the non-deterministic *choose strategy* available from the execution factory at the clock’s locus (see *fUML Specification,* Subclause 8.2.1). Each time the *advanceTime* operation is called, it decrements the wait duration, only actually advancing the clock time when the wait duration reaches zero, at which point it again non-deterministically resets the wait duration for the next clock step.

Since the fUML semantics allows any strategy to be used for a non-deterministic choice (so long as a choice is made in the required range), a client of a clock cannot assume any fixed relationship between a single clock step and the actual duration of that step relative to the underlying flow of time. Note, however, that, if the maximum step duration is set to 1, then a clock step will, of necessity, always have a duration of exactly 1. Therefore, if it is desired to model a situation in which clocks at different loci are all synchronized, all such clocks should be given a maximum clock step of 1. In this case, the clocks will all directly measure the underlying base flow of time, and, therefore, all be synchronized with each other.

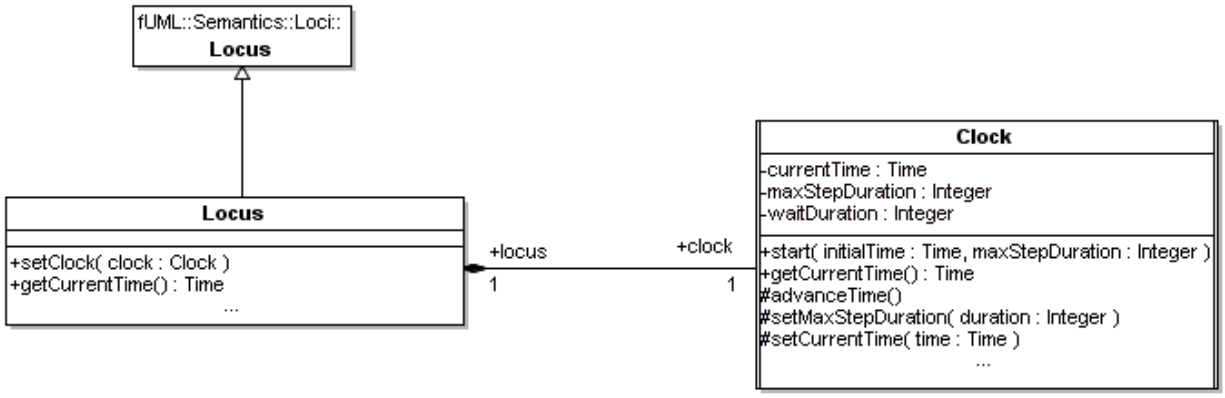

Figure 4-3 Clocks

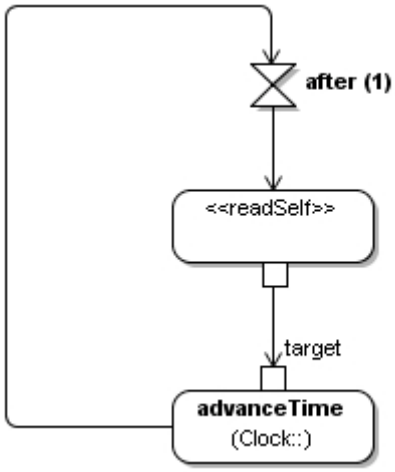

Figure 4-4 Clock Behavior

Threads

Beyond obtaining the current time, it is also necessary to provide the ability to *wait* until a certain time is reached. This capability is provided through an explicit model of *threads.* As shown in Figure 4-5, a thread is an abstract of a flow of execution that may be *suspended* and then *resumed* at a predetermined time.

When a thread is suspended at a locus, its suspension is registered with the clock at that locus, along with the desired resumption time. When the clock time is advanced, the *advanceTime*

operation checks if there are any threads whose resumption time has been reached. Any such threads are then removed from the suspension list and their *resume* operations invoked.

A *release* operation is also provided on Locus (which simply calls the corresponding Clock::release operation), to allow a thread to be removed from the suspension without being resumed (e.g., when it is terminated before the resumption time).

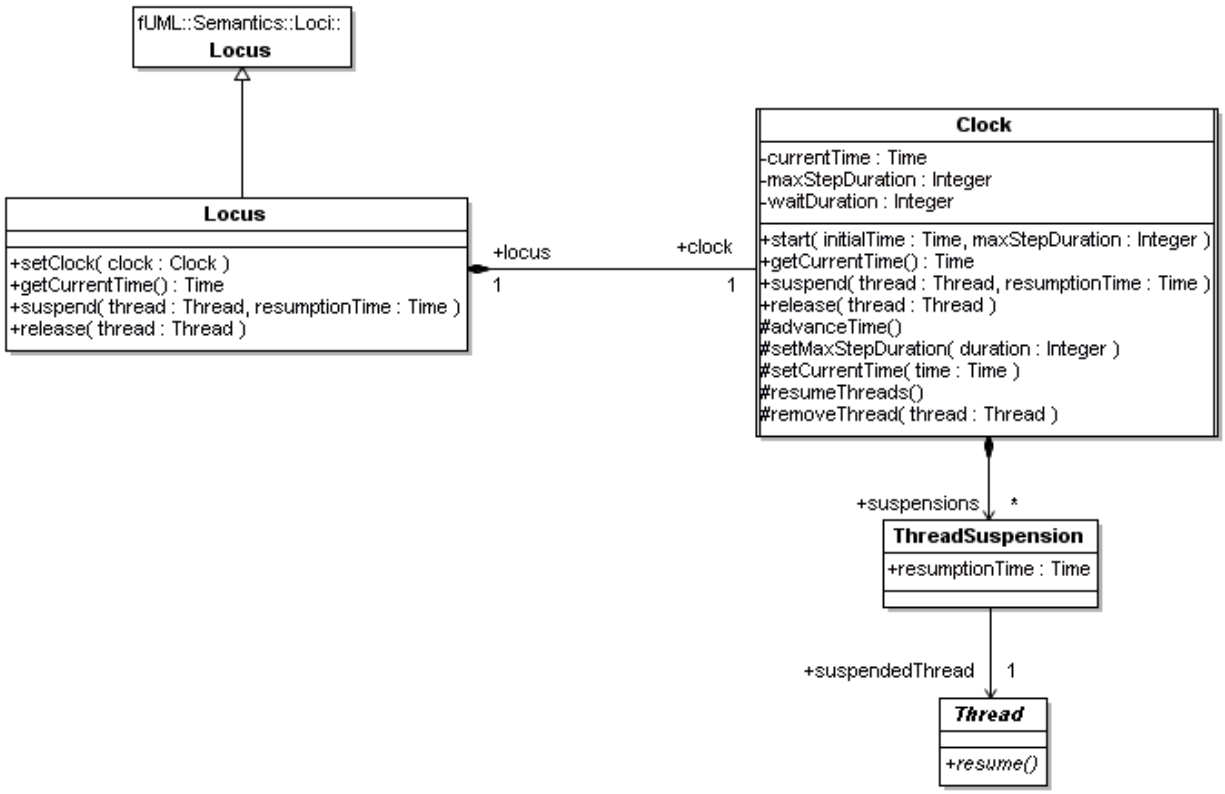

Figure 4-5 Threads

Semantic Visitors

The specification of the semantics for timing requires special handling for instantiating the semantics visitor classes for actions. As discussed in Section 4.3.3, the normal action activation visitor instance needs to be *wrapped* in an instance of the specialized UML4SysMLWithTime ActionActivation class. As shown in Figure 4-6, a specialized ExecutionFactory class is used to override the inherited *instantiateVistor* operation in order to provide this functionality. The specialized operation checks whether the given element is an action and, if so, wraps its action activation instance in an instance of the UML4SysMLWithTime ActionActivation class. Otherwise, it returns the original unwrapped visitor.

The only exception to the above specialized behavior is for AcceptEventAction. Duration constraints are not supported for accept event actions (see DurationConstraint in Section 4.1.2), because it is not clear what their semantics would be in that case. Therefore, the action activation for an accept event action is *not* wrapped as indicated above. However, the visitor returned is an instance of the specialized UML4SysMLWithTime AcceptEventActionActivation class (see Section 4.3.3), rather than the usual fUML version.

Note that the UML4SysMLWithTime ExecutionFactory specializes the UML4SysML ExecutionFactory discussed in Section 3.1.2.2. This means that, other than the specific UML4SysMLWithTime cases discussed above, semantic visitors are instantiated as required for the remaining UML4SysML semantics.

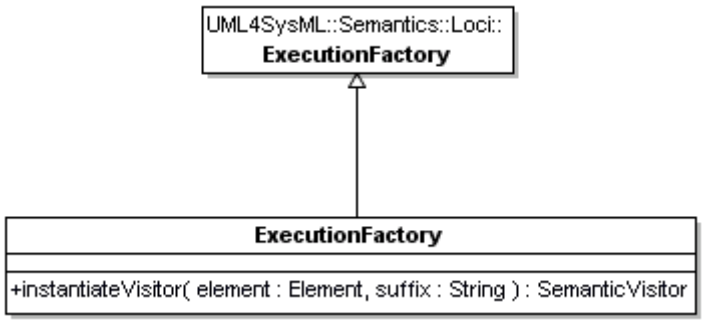

Figure 4-6 ExecutionFactory for UML4SysMLWithTime

4.3.2 Classes

The UML4SysMLWithTime Classes::Kernel package extends the fUML value model to provide the semantics of time values and the fUML evaluation model to provide for the evaluation of time expressions and durations.

Time Values

Since the Time type is a primitive type (see Section 4.2.1), its instances must be primitive values. As shown in Figure 4-7, it is therefore necessary to define a TimeValue subclass of the PrimitiveValue class. Time values are represented as the integer duration after the earliest time (which thus has the integer value 0). Note that, since durations are simply represented as non-negative integer values, no new value class is required to model their semantics.

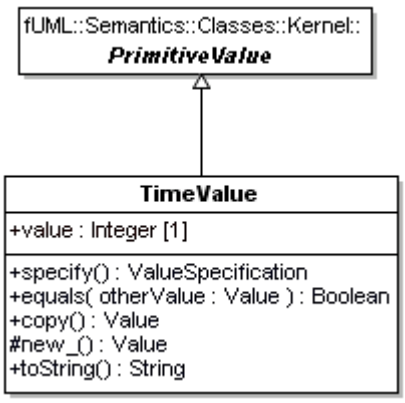

Figure 4-7 Time Values

Time Expression and Duration Evaluations

The SimpleTime abstract syntax package adds two new kinds of ValueSpecification: TimeExpression and Duration (see Section 4.1.1). Evaluating these requires two new

corresponding subclasses of Evaluation: TimeExpressionEvaluation and DurationEvaluation (see Figure 4-8).

Since time expressions are required to have an associated literal integer expression (see Section ·), the TimeExpressionEvaluation::evaluate operation proceeds by evaluating this literal integer. It then adds the resulting integer value to the current time to produce a time value that is the value of the time expression.

A duration is similarly required to have an associated literal integer expression (see Section ·). The DurationEvaluation::evaluate operation then proceeds by evaluating this literal integer, whose integer value is directly returned as the value of the duration.

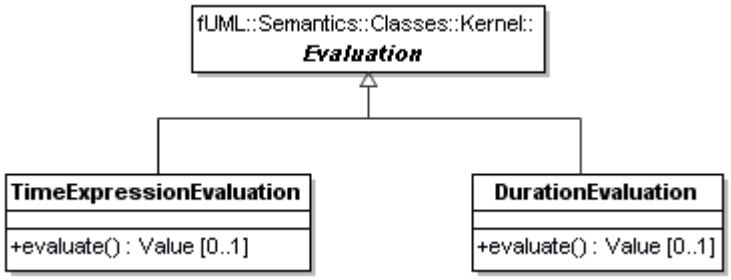

Figure 4-8 Time Expression and Duration Evaluations

4.3.3 Actions

Duration Constraints

The main reason for including the additional syntax from SimpleTime (see Section 4.1.2) is to allow duration constraints to be attached to actions within an activity. Such a constraint requires that each firing of an action have an elapsed time that is between the minimum and maximum durations given in the duration constraint. However, this leaves considerable leeway in the possible ways the action may execute while meeting this constraint.

The preliminary specification described here for action activation under a duration constraint takes a simple approach. When an action fires, carries out its required behavior as defined in the standard fUML execution model, with zero elapse time. This means that any interaction with other model execution state (such as reading or writing attributes) effectively occurs at the initial firing time of the action.

Once the behavior of the action has been completed, the execution of the action then delays for a period of time within the interval given in the duration constraint, before making offers on outgoing activity edges from the action. The actual delay is chosen non-deterministically between the minimum and maximum allowed durations, using the choice strategy at the execution locus for the action activation.

NOTE: To allow for specifically tailoring the execution of actions with duration constraints, without affecting other uses of the choice strategy, it would ultimately be better to have a separate kind of strategy for selecting a firing duration. The strategy used could then be fixed for a specific execution run to, for example, always choose the minimum duration or always use the maximum duration or to select a duration using some probability distribution between the

minimum and maximum. An even more sophisticated approach would provide some way to allow different specialized strategies to be used for different actions.

An issue with modeling the above semantics as an extension to the fUML execution model is that the new semantics apply to the execution of all types of actions. These means that one would wish to place the new behavior in the ActionActivation superclass. However, one also wants to be able to so specialize the ActionActivation class without having to also respecify all its concrete subclasses.

In order to achieve this, the UML4SysMLWithTime Actions::BasicActions defines a specialized ActionActivation class as a *wrapper* around the original fUML ActionActivation class (see Figure 4-9). Instead of directly using an instance of the normal fUML visitor class for actions, the usual instance is *wrapped* in an instance of the specialized ActionActivation, and it is that instance that is included in the containing activity node activation group. This specialized instantiation for action activations is handled by the specialized ExecutionFactory for UML4SysMLWithTime (see Section 4.3.1).

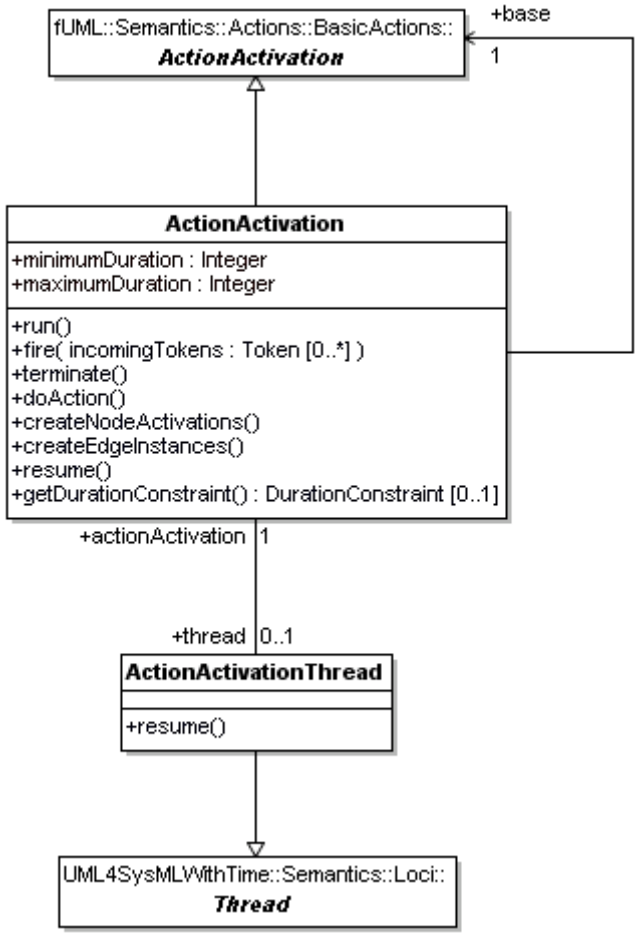

Figure 4-9 Action Activation with Duration Constraint

The specialized ActionActivation class overrides the following inherited operations.

- *run.* In addition to the normal initialization of the action activation wrapper, the specialized run operation also initializes the minimum and maximum duration attributes. If the action has no duration constraint, these are initialized to zero. Otherwise, they are set to the values obtained from evaluating the min and max durations of the duration constraint. If there is a duration constraint, then the run operation also creates an action activation thread to be used to suspend the execution of action for the appropriate amount of time. Finally, the specialized operation runs the base action activation.
- *fire.* The specialized fire operation first calls the specialized doAction operation for the wrapper. Then, if the action has a duration constraint, the fire operation non-deterministically computes a delay within the minimum and maximum allowed durations and suspends the action activation thread to a time equal to the current time plus the delay. If the action does not have a duration constraint, then the fire operation calls the resume operation, which simply carries out the remainder of the behavior found in the superclass fire operation (see below).
- *terminate.* In addition to terminating the wrapper, the specialized termination operation also terminates the base activation. Further, if there is an action activation thread, then this is released from the current execution locus, to cancel any possible pending resumption.
- *doAction.* The wrapper and base activations both have pin activations corresponding to all the pins of the action being executed (see *createNodeActivations* below), but it is the wrapper's pin activations that will actually be wired to activity edge instances in the context of the containing activity execution. Therefore, the doAction operation is specialized to first move any tokens on the input pin activations for the wrapper to the corresponding pin activations for the base and then call the doAction operation on the base.
- *createNodeActivations.* In addition to carrying out the usual action activation createNodeActivations behavior (that is, creating the pin activations for the wrapper) the specialized createNodeActivations operation initializes the node, running and group attributes of the base and then calls the createNodeActivations operation on the base (which will create corresponding pin activations on the base).
- *createEdgeInstances.* The createEdgeInstances operation is specialized to simply delegate to the base createEdgeInstances operation.

In addition, the specialized ActionActivation class adds two new operations.

- *resume.* If the action activation has been suspended, then, once its thread is resumed, the ActionActivationThread::resume operation calls the ActionActivation::resume operation. The ActionActivation::resume operation first moves any tokens from the output pin activations of the base action activation to the corresponding pin activations of the wrapper. It then completes the behavior of the superclass fire operation, sending offers and checking if the action activation should immediately fire again.
- *getDurationConstraint.* This operation returns the duration constraint, if any, applied to the action for this action activation. It does this by searching all the owned members of the activity containing the action for a duration constraint with the action as its constrainedElement.

NOTE:

The use of a thread to suspend the execution of an action activation will not actually work as desired. This is because, once the action activation thread has been suspended with the execution locus, the invocation of the fire call will return. In the implicit threading model of the fUML execution model (see, in particular, the discussion in *fUML Specification,* Subclause 8.5.2.1), this indicates that the thread of control through the action activation is either complete or blocked. When all such threads of control in an activity execution return, the execution terminates. Currently, this will happen regardless of whether some of the threads are actually suspended for later resumption.

The desired behavior would be for the activity execution itself to remain suspended and uncompleted while any of the threads within it are suspended. With the current implicit threading model, this is quite difficult to do, since there is no explicit model of all the threads within an activity execution, and the activity execution may itself be the result of an invocation in an implicit thread of control within some calling activity execution. Rather than attempting a solution to this problem during the present project, an issue will be raised with the fUML FTF requesting consideration of revising the fUML execution model to explicitly model threads in general. This will allow for a more tractable extension in the future to accommodate the semantics of delays for duration constraints.

Time Events

The UML4SysMLWithTime Actions::CompleteActions package contains a specialization of the fUML AcceptEventActionActivation class and a concrete AcceptEventActionThread subclass of Thread (see Figure 4-10). The specialized AcceptEventActionActivation class handles the semantics of an accept event action triggered by a time event. It overrides the following inherited operations.

- *run.* The specialized run operation checks whether the action for this activation has a trigger with a time event. If so, the operation creates an accept event action thread to be used to suspend the accept action activation when triggered.
- *fire.* If the accept event action has a time event trigger, the specialized fire operation computes the resumption time by evaluating the time expression of the time event and suspends the thread associated with this activation until that time. Otherwise, the functionality is the same as the fUML AcceptEventActionActivation fire operation.
- *terminate.* In addition to the functionality of the fUML AcceptEventActionActivation terminate operation, the specialized operation releases the accept event action thread for this activation, if there is one. This cancels any suspension the thread might have with the execution locus.

The specialized AcceptEventActionActivation class also defines the following additional operation.

- *resume.* If the action activation has been suspended based on a time event, then, once its thread is resumed, the AcceptEventActionThread::resume operation calls the AcceptEventActionActivation::resume operation. The AcceptEventActionActivation::resume operation then completes the firing of the accept event action activation, by sending any offers and checking whether the activation should immediately fire again.

As mentioned in Section 4.3.1, a UML4SysMLWithTime accept event action activation is *not* wrapped in a UML4SysMLWithTime action activation. This is because an accept event action is

simply a specification of a response to an event occurrence, and it is not clear what meaning a duration constraint would have for this. Therefore, duration constraints are not allowed on accept event actions (see DurationConstraint in Section 4.1.2).

NOTE: Unlike the problem with threading for handling the semantics of duration constraints (as discussed above), the use of a thread to suspend an accept event action activation does work as desired. This is because fUML only allows threads in classifier behaviors or standalone activities, not in synchronously invoked operation methods. Suspending an accept event action activation with a thread works analogously to the way that such an activation is suspended using an accept event action event accepter in the regular fUML execution model (see *fUML Specification,* Section 8.6.4.1).

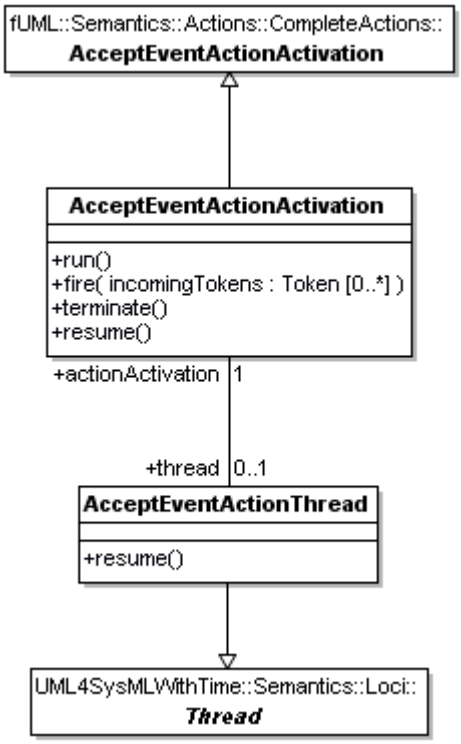

Figure 4-10 AcceptEventActionActivation for UML4SysMLWithTime

4.3.4 Class Descriptions

See the UML4SysMLWithTime semantic model file for the descriptions of the above classes and Java code specifications for all operations.

Section 5 Structured Model Execution

According to the project SOW, the Structured Model Execution task is required to:

> *"Specify the basic semantics of blocks, parts, ports and connectors and the semantics for the realization of the actions in activity partitions by the associated blocks and parts that are represented by the activity partitions. In particular, define the required semantics of standard ports and flow ports and how the behavior of a part is expressed to be consistent with the behavior of the activity diagram."*

Semantics for this area were not completed within the period of performance for the project. However, one topic important to this area (and SysML in general) is how to specify the semantic effect of stereotypes. In the case of structured models, relevant stereotypes include those for blocks, flow ports, allocation to partitions, etc. Considerations relevant to this topic are documented in this section.

5.1 Stereotype Semantics

5.1.1 Abstract Syntax

The fUML execution model specifies the execution of a model represented in the abstract syntax subset defined for fUML. The first issue that must be addressed in extending the execution model to handle stereotypes is that the *UML Superstructure* does not actually provide an abstract syntax model for the application of a stereotype. Rather, Subclause 18.3.2 defines the semantics of stereotype extension in terms of an "equivalence to a MOF construction".

For example, consider the definition of the SysML Block stereotype, reproduced in Figure 5-1. This has the "MOF equivalent" model shown in Figure 5-2. In this model, the Block stereotype is replaced with a Block metaclass and the extension relationship is replaced with an association to the UML4SysML::Class metaclass.

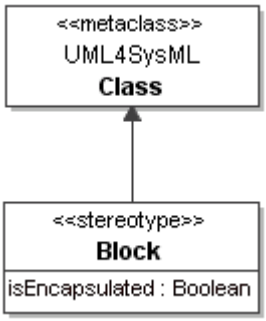

Figure 5-1 Block Stereotype

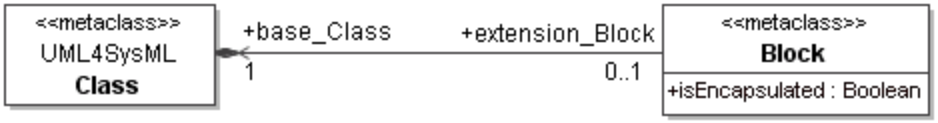

Figure 5-2 MOF Equivalent Model for the Block Stereotype

Note that the association in Figure 5-2 is a composition association, since, when a class stereotyped as a block is deleted from a model, its applied block stereotype instance should be removed, too. However, unusually for a composition association, the association is navigable only from the extension to the base (composite), *not* in the other direction. This is because, in the UML metamodel, navigable association ends are always considered to be classifier owned. Thus, if the association was navigable from the base to the extension, the implication would be that adding the stereotype Block would require a modification to the UML metamodel in order to add a property for the extension_Block association end to the metaclass Class—which would be against the design goal for the UML profile mechanism to allow "lightweight" metamodel extensions.

So, consider now the application of the Block stereotype shown in Figure 5-3. According to the MOF equivalent of Figure 5-2, this stereotype application corresponds to the instance model shown in Figure 5-4. Note again the unidirectionality of the link *from* the Block instance *to* the Class instance.

Figure 5-3 Example Stereotype Application

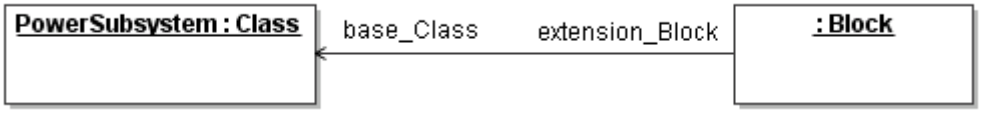

Figure 5-4 MOF Equivalent Model for Stereotype Application

Now, in full UML 2, it is still generally possible to access the value of a non-navigable end of an association. For example, given just the class instance for PowerSubsystem, one could get the extent of all Block instances that currently exist (using, say, a read extent action) and then search this collection for the instance for which the base_Class is PowerSubsystem. This may not be as "efficient" as direct navigation across the link to the extension_Block end, but it is in principle possible.

However, for the purposes of extending the fUML execution model to account for the semantics of blocks, the models in Figure 5-2 and Figure 5-4 must be interpreted in terms of the Base UML (bUML) subset in which the fUML execution model is written (see Clause 10 and Annex A of the *fUML Specification*)—and the bUML subset does not include read extent actions. Thus, in bUML, the MOF equivalent model of Figure 5-2 provides, in principle, no way to determine for a class in a user model whether it has a Block stereotype applied (or any other stereotype for that matter).

The proposed approach to deal with this issue is to incorporate stereotypes into the UML4SySML abstract syntax using the standard MOF equivalent model *except* that the composition associations are all *bidirectionally* navigable. This *Extended UML4SysML* model would be a standard MOF metamodel providing a representation for the SysML abstract syntax that is different from, but equivalent to, the UML4SysML metamodel plus SysML profile

representation used in the *SysML Specification.* An extended execution model for SysML could then operate on complete SysML user models represented using instances of the Extended UML4SysML metaclasses, including all stereotype applications.

5.1.2 Execution Model

Once the syntactic issues discussed in Section 5.1.1 are resolved, it is then necessary to actually extend the fUML execution model to actually provide operational semantics for SysML stereotypes. Note that this goes beyond the extensions discussed in Section 3 and Section 4, which dealt with providing semantics for features in UML4SysML that are not in the fUML subset, but which are still standard UML 2 features. The SysML stereotypes, on the other hand, denote semantics that are specific to SysML.

Of course, the simplest way to model these SysML semantics is to still use basically the same techniques as in Section 3 and Section 4. That is, consider the Extended UML4SysML metamodel to be just a further extension of the fUML abstract syntax subset: from fUML to UML4SysML (in Section 3) to UML4SysMLWithTime (in Section 4) to Extended UML4SysML (here).

In this approach, one could, for example, create, say, an ExtendedUML4SysML Object semantic visitor class specializing the base UML4SysML Object class, which would be instantiated as the visitor of the ExtendedUML4SysML Class syntax class. As discussed in Section 5.1.1, in the Extended UML4SysML abstract syntax model, Class has an association to a Block metaclass representing the properties of an (optionally) applied Block stereotype. The ExtendedUML4SysML Object class would then check for whether a Block instance was actually applied to its class and, if so, model the appropriate semantics. If not, then it would simply inherit the standard UML base object semantics.

In principle, all SysML semantics could be included in the execution model in this way, effectively treating SysML as entirely its own modeling language with the Extended UML4SysML abstract syntax, rather than as a profile. However, it would seem to be desirable to have a more general way to handle profile extensions and stereotype application than to have a hand-tailored execution model for each profile. In particular, with this approach, it would seem that, in order to specify the semantics of applying multiple profiles, one would need to craft a specialized execution model for each possible combination of applied profiles!

In effect, UML profile mechanism provides a compositional method for attaching syntactic annotations to UML model elements (via MOF equivalent models similar to that shown in Figure 5-2), and one would like to have a similarly compositional approach for specifying the added semantics indicated by these annotations. One possible way to address both these problems is to use a *wrapping* approach similar to that used for ActionActivation in UML4SysMLWithTime in order to handle duration constraints (see Section 4.3.3). In this approach, the Extended UML4SysML Object class would still specialize the base UML4SysML Object class, but it would also have a reference to an instance of this base class. Rather than inheriting non-SysML specific behavior from the base Object class, such behavior would be delegated to the base instance.

Using this approach, a single base semantic visitor instance could have multiple wrappers representing multiple applied stereotypes from different profiles. The simplest way to do this would be to apply the wrappers sequentially—that is, the wrapper for each successive stereotype

would have as its base instance a semantic visitor already wrapped for all previous stereotypes. Semantic behavior would then be sequentially delegated from the outermost wrapper down to the appropriate wrapper level (or original base level) at which it should be handled.

If the required semantics added by each of the applied stereotypes were mutually independent, then the sequential ordering would not matter. However, if there are interrelationships between the semantics of various applied stereotypes, then the specified behavior could differ depending on the order in which the wrappers are applied, since inner wrappers have know knowledge of what wrappers may be applied in outer layers. In this case, it may be desirable to have multiple wrappers all pointing together to a single base semantic visitor instance, with some sort of compositional coordinator to pull them together. This is still a topic for future research, though.

Unfortunately, there is also a more fundamental problem with applying this semantic wrapper approach to the current fUML execution model. The classes in the fUML execution model intentionally have visible attributes, in order to simplify both the class diagrams for the model and the specification of behavior for operations on semantic classes. Reading and writing of these attributes are mapped directly to UML read and write structural feature actions.

As a result of this, though, it is not possible, in general, to keep the values of the attributes of a wrapper instance synchronized with the values of the attributes of its base instance. When the wrapper delegates behavior to the base instance, that instance will update its own attributes, perhaps making further calls out to other objects that will then need to access those attribute values. However, external references will always be back through the wrapper instance, whose attributes will not have been yet appropriately updated.

The standard way to handle this is to require that all attribute access be through getter and setter operations. That way, get and set calls to the wrapper can be delegated to the base instance, just as can be done for other sorts of operations. Adding getter and setter operations to all classes in the fUML execution model, though, would significantly clutter the model without adding anything in particular for the fUML specification itself, which does not currently deal with profiles at all.

Nevertheless, the ability to handle profile semantics may be important enough that it is worth updating the *fUML Specification* to better accommodate it. This will be suggested to the fUML Finalization Task Force for consideration (as listed in Section 8.1). Pending the acceptance of some basic common philosophy for building profile semantics on the fUML foundation, semantics extensions for the SysML profile were not detailed further in this project.

Section 6 Formal Operational Semantics

According to the project SOW, the Formal Operational Semantics task is required to:

> *"Develop a grounding for the operational semantic specifications created in 1-4 above, using appropriate formal or semi-formal techniques. This grounding should extend and encompass the base semantics required as part of the Foundational UML submission…."*

In extending the fUML semantics to cover SysML, it will be important to ensure that the formal base semantics continues to provide sufficient grounding to cover the semantic extensions. It is therefore relevant to further explore and explicate the semantic grounding of the current *fUML Specification*. This was the topic of discussion at the Executable UML/SysML Semantics Project Meeting held on 01 August 2008.

Clause 10 of the *fUML Specification* defines the *base semantics* for Foundational UML (fUML). This base semantics is actually provided for the further subset of Base UML (bUML) that is used in writing the execution model that provides the operational semantics for all of fUML (see Clause 8 of the *fUML Specification*). The specification approach is based on using *first order logic* to specify the relationship between UML syntactic elements and instances in the semantic domain.

Section 6.1 includes a description of the base semantic approach used in the *fUML Specification* and its relation to traditional concepts of mathematical logic, based on the discussion held at the August project meeting. This can be considered to be an elaboration of description of background concepts in Clause 6 of the *fUML Specification*.

The semantic specification for timing presented in Section 4 requires one addition to the base semantics provided in the *fUML Specification:* the base semantics of time events. Section 6.2 includes a brief discussion of how this might be added.

6.1 Model Semantics and Mathematical Logic

Most of the semantic specification in Clause 10 of the *fUML Specification* has to do with the semantics of behavioral modeling, particularly activities. However, the fundamental concepts involved in the use of mathematical logic for semantic specification can be largely understood by just considering the simpler semantics of structural modeling. Therefore, for simplicity of presentation, the discussion here will focus only on structural semantics.

6.1.1 Modeling

Consider the simple class model shown in Figure 6-1. This is intended to be interpreted as allowing certain statements to be made about the domain being modeled: "A person has a name." "A person owns houses." And so on.

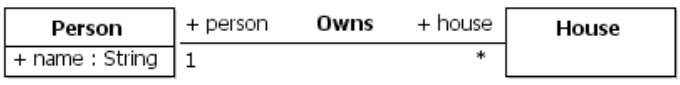

Figure 6-1 Simple Class Model

These informal English statements can be formalized using mathematical logic. First, we need to define a set of *predicates* for the basic terms specified in the model. These include:

- Person(p) means "p is a Person".
- name(p,n) means "p has the name n".
- House(h) means "h is a house".
- Owns(p,h) means "p owns h".[4]

We can than formally interpret statements made by the model in Figure 6-1 using the typical notation of mathematical logic:

- Property *name* is an attribute of class *Person*: "A name is for a person."
  $\forall p(\text{name}(p,n)\rightarrow\text{Person}(p))$
- Property *name* has type *String*: "The name of a person is a string."
  $\forall p\forall n(\text{name}(p,n)\rightarrow\text{String}(n))$[5]
- The multiplicity of *Person::name* is 1..1: "Every person has exactly one name."
  $\forall p(\text{Person}(p)\rightarrow\exists n(\text{name}(p,n)\wedge\forall m(n\neq m\rightarrow\neg\text{name}(p,m))))$
- The association *Owns* has end types *Person* and *House*: "Persons own houses".
  $\forall p\forall h(\text{Owns}(p,h)\rightarrow\text{Person}(p)\wedge\text{House}(h))$
- The multiplicity of *Owns::house* is 1..1: "Every house has exactly one owner."
  $\forall h(\text{House}(h)\rightarrow\exists p(\text{Owns}(p,h)\wedge\forall q(p\neq q\rightarrow\neg\text{Owns}(q,h))))$

The fUML Specification does not actually use the traditional mathematical logic symbology used above. Instead, it expresses logical statements using the Common Logic Interchange Format (CLIF).[6] In this machine-readable format, atomic predicate formulas such as "Person(p)" are written in the form "`(Person p)`". The above statements are then written in CLIF as:

(1) "A name is for a person."

```
(forall (p b)
    (if (name p n) (Person p)))
```

(2) "The name of a person is a string."

```
(forall (p n)
    (if (and (Person p) (name p n)) (String n)))
```

(3) "Every person has exactly one name."

```
(forall (p)
    (if (Person p) (exists (n) (name p n))))
```

[4] Note that we use a two place predicate to represent an association (e.g., "Owns(p,h)"), rather than using two separate predicates for the association end properties (e.g., "person(p,h)" and "house(h,p)"). Even though Clause 10 of the fUML specification uses the latter formalization in terms of association ends, treating them as multi-argument predicates is more consistent with normal practice in logic and simpler for our purposes here. And it does not affect the relevance of the points being made here to the actual approach in the fUML specification.

[5] We are assuming here that the primitive concept of a "String" is pre-defined and that "String(s)" means "s is a String".

[6] See ISO 24707, http://standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007(E).zip.

(4) "Persons own houses."

```
(forall (p h)
    (if (Owns p h) (and (Person p) (House h))))
```

(5) "Every house has exactly one owner."

```
(forall (h)
    (if (House h)
        (exists (p)
            (and (Owns p h))
                 (forall (q) (if (not (= p q)) (not (Owns q h))))))))
```

Now consider the model in Figure 6-2 of instances of classes from Figure 6-1. This model also makes statements about the domain being modeled, but it now makes statements about *specific things* in that domain. The set of such things is what logicians call the *universe of discourse*—in this case, people and houses.

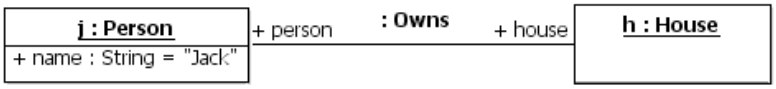

Figure 6-2 A Simple Instance Model

Just as we could formally interpret that statements made by the class model of Figure 6-1, we can make formal interpretations of the statements made by the instance model. Such statements have the form of *assertions* about specific elements of the universe of discourse—in this case, about the person labeled *j* in Figure 6-2 and the house labeled *h.* An assertion is a statement that is claimed to be true for the universe of discourse.

The assertions made by the instance model in Figure 6-2 include the following:

(6) "*j* is a person."

```
(Person j)
```

(7) "The name of *j* is Jack."

```
(name j "Jack")
```
[7]

(8) "*h* is a house."

```
(House h)
```

(9) "*j* owns *h.*"

```
(Owns j h)
```

6.1.2 Deduction, Syntax and Semantics

The primary advantage of formalizing the statement made by a model as logical propositions is that the rules of logic may be used to *deduce* new statements from statements that are already known. Such deductions are made according to *deduction rules* that are carefully formulated so that any deduced statements are true as long as all the statements they are deduced from are true.

Some common deduction rules are:

(10) Given `(and P Q)`, deduce `P` and `Q`.

[7] Note that we are assuming here that there is a built-in notation for strings like "Jack".

(11) Given `(if P Q)` and `P`, deduce `Q`.

(12) Given `(if (and P1 P2) Q)` and `P1`, deduce `(if P2 Q)`.

(13) Given `(forall (x) (P x))`, deduce `(P y)` for any `y`.

As an example deduction, consider that, given statement (5) from the class model and statement (9) from the instance model, we can use rules (13) and (11) to deduce:

(14) "No one other than *j* owns *h.*"
```
(forall (q) (if (not (= j q)) (not (Owns q h))
```

Now, suppose further that we know of another person *k* who is not *j*:

(15) `(Person k)`

(16) `(not (= j k))`

Then, using rules (13) and (11) again, we can deduce `(not (Owns b h))`.

Deduction can be used not only to derive new true statements, but also to check on the consistency of a set of assertions. For example, from statements (4) and (9) we can deduce `(Person j)` and `(House h)`, using rules (10), (11) and (13). This is consistent with assertions (6) and (8) already made by the instance model.

In the parlance of logic, what we are doing here is taking the statements made by a model, as formalized in mathematical logic, to be a set of *axioms*—that is, a set of statements all asserted to be true together. The (generally infinite) set of all possible deductions that can be made from these axioms using some set of deduction rules is known as the *theory* derived from the axioms.[8] A *proof* of a statement in the theory is a series of deductions leading from the axioms to the statement. A theory is *consistent* if there is no statement `S` in the theory for which the logical inverse of that statement `(not S)` is also in the theory (that is, it is not possible to prove both the statement and its inverse in the theory).

In *first order* logic, statements are written in terms of predicates over some universe of discourse and all quantification ("`forall`" and "`exists`") is over this universe. In Section 0 we interpreted UML class and instance models in terms of such first order logic, in which each model implied certain predicates and certain axiomatic statements in terms of those predicates.

However, there are some general statements we would like to make about models such as those discussed in Section 0 that cannot be handled by our formal language so far. For example, in general in UML, if a property has a certain type, then all values of that property must be of the given type. Since Figure 6-1 shows that the property *Person::name* has the type *String,* we would like to be able to *deduce,* from the general rules of UML, that any value of *name* must be a string.

In order to formalize such general rules, we need to make statements about elements of the models themselves, such as classes like *Person* and properties like *name.* Since we interpreted these elements as logical predicates, we thus need to make formal statements about *predicates.* A

[8] Clause 6 of the fUML Specification defines a "theory" as the set of deduction rules used, rather than the statements deduced using those rules. This is closer to the way the term *theory* is used outside of mathematical logic, but the conventional definition within mathematical logic is the one given here.

logic language that allows formal statements to be made about first order predicates is known as *second order.*

For example, we can define the following second order predicates, whose arguments are first order predicates over the original universe of discourse:

(17) "*C* has the owned attribute *A*."
```
(ownedAttribute C A)
```

(18) "*P* has the type *T*."
```
(type P T)
```

We can now state the general axioms:

(19) "If *A* is an owned attribute of *C* and *y* is a value of *A* for instance *x,* then *x* is an instance of *C*."
```
(forall (C A x y)
    (if (and (ownedAttribute C A) (A x y)) (C x)))
```

(20) "If *A* is an owned attribute of *C*, *A* has type *T, x* is an instance of *C* and *y* is a value of *A* for *x,* then *x* is an instance of *T*."
```
(forall (C A T x y)
    (if (and (ownedAttribute C A) (type A T))
             (C x) (A x y))
        (T y)))
```

If we then assert

(21) `(ownedAttribute Person name)`

(22) `(type name String)`

we can *deduce*

(23) `(forall (x y) (if (name x y) (Person x)))`

(24) `(forall (x y) (if (and (Person x) (name x y)) (String y)))`

which we previously simply *asserted* for the model as (1) and (2).

Now, second order languages are generally more difficult to deal with than first order languages, and automated provers, for example, generally only handle first order languages. To deal with this, logicians often recast a second order language into an equivalent first order form.[9] To do this, we extend our universe of discourse with new elements to represent what were formerly the first order predicates of the language. What were formerly second order predicates then become first order predicates over these new elements.

For the purposes of UML modeling, we need to add a set of *class* elements to represent classes, a set of *property* elements to represent properties, etc. We then define new predicates to distinguish these elements from the instances that were already in the universe:

[9] Technically, it can be shown that the direct or *absolute* semantics for a second order language cannot be fully duplicated by any first order language. However, there is an *alternate* semantics for second order languages that is essentially first order and which is equivalent for our purposes here. (See, e.g., Herbert B. Enderton, *A Mathematical Introduction to Logic, Second Edition,* Academic Press, 2001, Chapter 4.)

(25) "*c* is a class."
```
(Class c)
```

(26) "*t* is a primitive type."
```
(PrimitiveType t)
```

(27) "*p* is a property."
```
(Property p)
```

(28) "*a* is an association"
```
(Association a)
```

Thus, for the sample model of Figure 6-1, we have

(29) `(Class Person)`

(30) `(Class House)`

(31) `(PrimitiveType String)`

(32) `(Property name)`

(33) `(Association Owns)`

Finally, we need to define some additional predicates to be used in place of applying the old first-order predicates:

(34) "*c* classifies *v*."
```
(classifies c v)
```

(35) "*p* has value *v* for *x*."
```
(property-value x p v)
```

(36) "*a* links *x* and *y*."
```
(link x a y)
```

for which the following basic axioms hold

(37) `(forall (p) (if (PrimitiveType p) (Classifier p)))`

(38) `(forall (c) (if (Class c) (Classifier c))`

(39) `(forall (c v) (if (classifies c v) (Classifier c)))`

(40) `(forall (x p v) (if (property-value x p v) (Property p)))`

(41) `(forall (x a y) (if (link x a y) (Association a))`

Thus, given (29) through (33), instead of `(Person p)`, `(name p n)` and `(Owns p h)`, we can now write `(classifies Person p)`, `(property-value p name n)` and `(link p Owns h)`. And this, in turn, lets us rewrite general rules such as (19) and (20) in a first order form:

(42)
```
(forall (c p x y)
    (if (and (ownedAttribute c p) (property-value x p y))
        (classifies c x)))
```

(43)
```
(forall (c p t x y)
    (if (and (ownedAttribute c p) (type p t)
             (classifies c x) (property-value x p y))
        (classifies t y)))
```

Then, given (21) and (22), we can make the equivalent deductions to (23) and (24):

(44) `(forall (x y) (if (property-value x name y) (classifies Person x)))`

(45)
```
(forall (x y)
    (if (and (classifies Person p) (property-value name p n))
        (classifies String n)))
```

Finally, rather than interpreting the instance model in Figure 6-2 in terms of the assertions (6) through (9), we have, instead, the equivalent assertions:

(46) `(classifies Person j)`

(47) `(property-value j name "Jack")`

(48) `(classifies House h)`

(49) `(link j Owns h)`

Then, given (44) and (45) (and other model-specific facts deduced from the general rules of UML), we can make deductions about the instance model, as discussed at the beginning of this section.

We can summarize the results of the discussion above by organizing our logic language into three categories:

- *Syntax* – Predicates such as `Class`, `Property` and `ownedAttribute` represent the *syntax* of UML models. A UML model (such as the class model in Figure 6-1) can be interpreted as a set of assertions using these predicates (such as (29) through (33), (21) and (22)).
- *Semantics* – Predicates such as `classifies`, `property-value` and `link` provide the basis for the *semantics* of UML models. They define how elements of a UML model can be used to make statements about instances in the domain being modeled. General rules such as (42) and (43) then specify how the structure of the model further constraints the valid statements that can be made about the domain.
- *Theorems* – From the general semantic rules, together with the syntactic assertions for a specific UML model, one can deduce model-specific *theorems* such as (44) and (45). These theorems, along with domain-specific assertions such as (46) through (49), can then be used to make specific deductions about the domain being modeled.

The abstract syntax of UML is, of course, determined by its specification (which we consider in a bit more detail below). It is the semantics of the bUML subset of UML, using essentially the approach discussed here, that is provided by Clause 10 of the fUML Specification.

6.1.3 Metamodeling

Consider the fragment of the UML abstract syntax model shown in Figure 6-3, which is sufficient to cover the syntax used in Figure 6-1. Since this is just a UML model, we can interpret it as a set of logical assertions, as discussed in Section 6.1.1. The result is a set of predicates `Class`, `Association`, `Property`, etc., that are exactly the syntactic predicates required in Section 6.1.2.

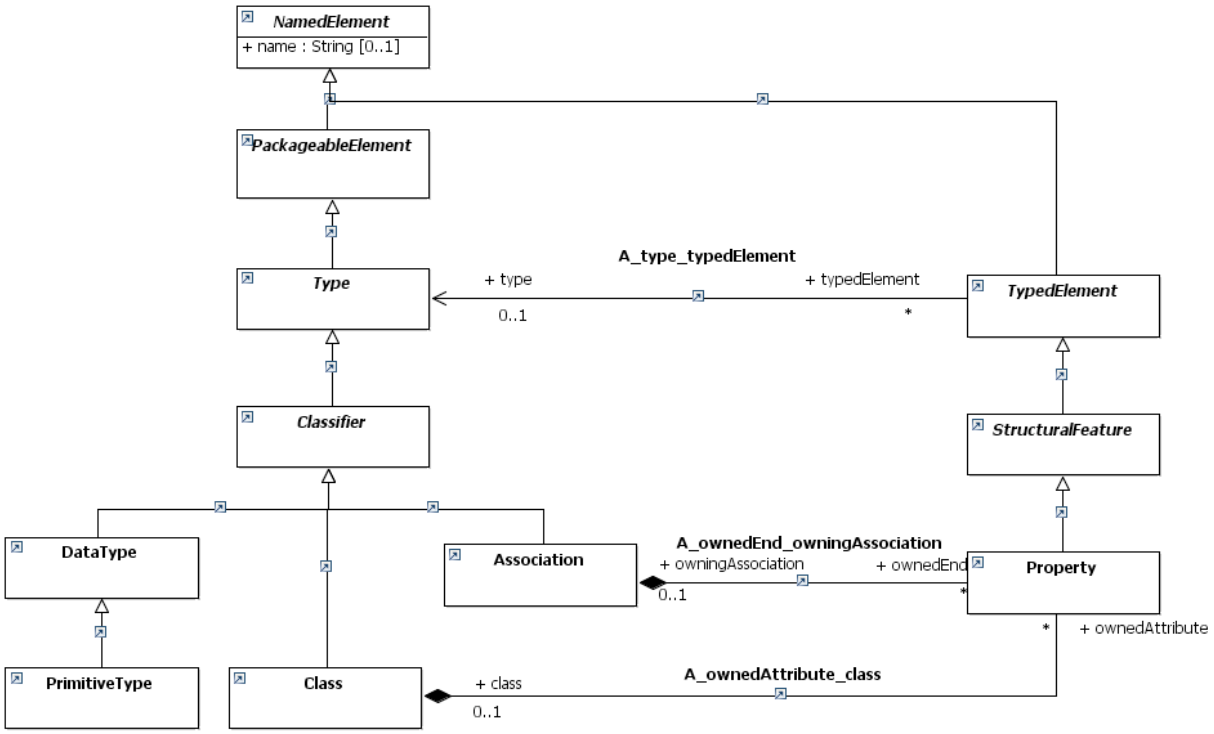

Figure 6-3 Classifier Abstract Syntax

In addition, the abstract syntax model places additional constraints that must be satisfied by any syntactically well-formed model. These include the following.

Generalizations:

(50) `(forall (r) (if (PackageableElement e) (NamedElement e)))`

(51) `(forall (t) (if (Type t) (PackageableElement t)))`

(52) `(forall (c) (if (Classifier c) (Type c)))`

(53) `(forall (c) (if (Class c) (Classifier c)))`

(54) `(forall (a) (if (Association a) (Classifier a)))`

(55) `(forall (d) (if (DataType d) (Classifier d)))`

(56) `(forall (p) (if (PrimitiveType p) (DataType p)))`

(57) `(forall (s) (if (StructuralFeature s) (TypedElement s)))`

(58) `(forall (p) (if (Property p) (StructuralFeature p)))`

Attributes:

(59) `(forall (n s) (if (name n s) (NamedElement n)))`

(60) `(forall (n s) (if (name n s) (String s)))`

(61)
```
(forall (n s1 s2)
    (if (and (name n s1) (not (= s1 s2))) (not (name n s2))))
```

Associations:

(62)
```
(forall (p c)
    (if (A_ownedAttribute_class p c) (and (Property p) (Class c))))
```

(63)
```
(forall (p c1 c2)
    (if (and (A_ownedAttribute_class p c1) (not (= c1 c2)))
        (not (A_ownedAttribute_class p c2))))
```

(64)
```
(forall (p a)
    (if (A_ownedEnd_owningAssociation p a)
        (and (Property p) (Association a))))
```

(65)
```
(forall (p a1 a2)
    (if (and (A_ownedEnd_owningAssociation p a1) (not (= a1 a2)))
        (not (A_ownedEnd_owningAssociation p a2))))
```

(66)
```
(forall (t e)
    (if (A_type_typedElement t e) (and (Type t) (TypedElement e))))
```

(67)
```
(forall (t1 t2 e)
    (if (and (A_type_typedElement t1 e) (not (= t1 t2)))
        (not (A_type_typedElement t2 e))))
```

For example, Figure 6-4 gives the abstract syntax representation of the simple model from Figure 6-1. This corresponds to the logical assertions:

(68) `(PrimitiveType s)`

(69) `(name s "String")`

(70) `(Property p)`

(71) `(name p "name")`

(72) `(A_type_typedElement s p)`

(73) `(Class c1)`

(74) `(name c1 "Person")`

(75) `(A_ownedAttribute_class p c1)`

(76) `(Property e1)`

(77) `(name e1 "owner")`

(78) `(A_type_typedElement c1 e1)`

(79) `(Class c2)`

(80) `(name c2 "House")`

(81) `(Property e2)`

(82) `(name e2 "houses")`

(83) `(A_type_TypedElement c2 e2)`

(84) `(Association a)`

(85) `(name a "Owns")`

(86) `(A_ownedEnd_owningAssociation e1 a)`

(87) `(A_ownedEnd_owningAssociation e2 a)`

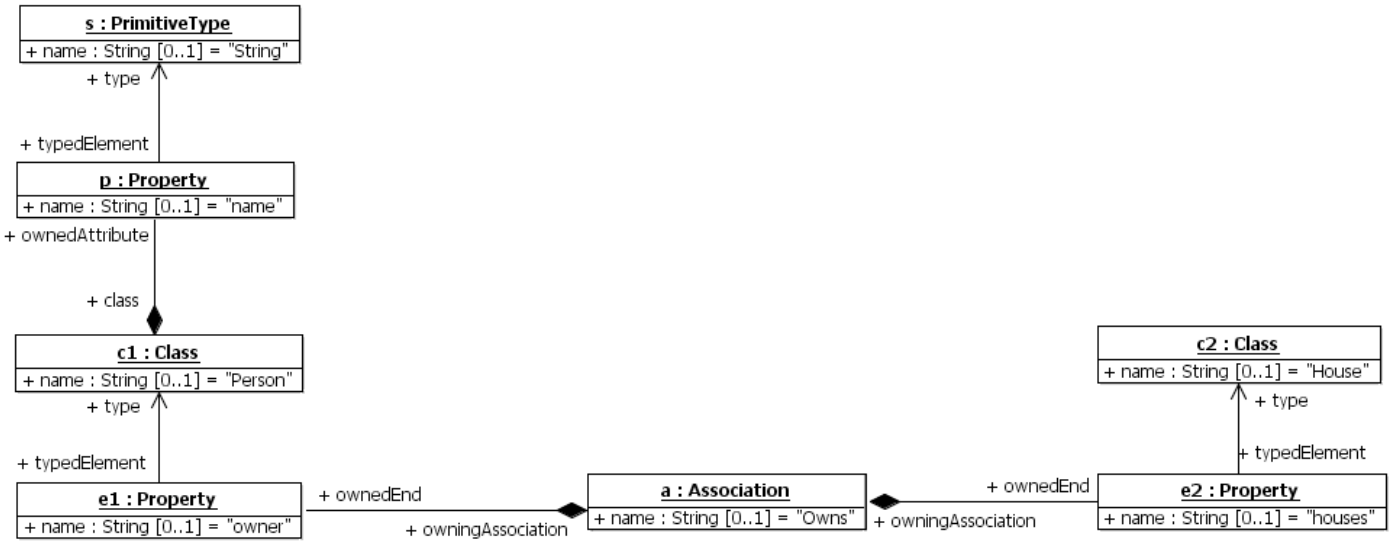

Figure 6-4 The Abstract Syntax Representation of the Model of Figure 6-1

The bUML semantics specified in Clause 10 of the fUML Specification presumes that a model has been represented in a logical form similar to this. Further, while the clause does not state these conditions explicitly, it is assumed that any model being interpreted satisfies all the logical well-formedness conditions derivable from the abstract syntax subset for bUML (such as (50) through (67)).

Now, there is, of course, also an abstract syntax for UML instance models. For example, the abstract syntax fragment given in Figure 6-5 is sufficient to cover the syntax used in the simple model of Figure 6-2. Interpreting this model leads to logical predicates such as `(A_classifier_instanceSpecification c i)`, `(A_value_owningSlot v s)` and `(A_slot_owningInstance s i)`. However, It is important to distinguish these from the seemingly similar predicates `(classifies c v)` and `(property-value x p v)`.

The statement `(A_classifier_instanceSpecification c i)` is an assertion of a *syntactic* relationship between two model elements: a classifier *c* and an instance specification *i.* On the other hand, `(classifies c v)` is an assertion of a *semantic* relationship between the classifier model element *c* and an actual instance *v* in the universe of discourse, giving meaning to what a "classifier" *is*. In terms of the OMG hierarchy of four meta-layers, `A_classifier_instanceSpecification` is a relation between two elements in layer M1 (*c* and *i*), while `classifies` relates an element in M1 (*c*) to an actual instance in M0 (*v*).

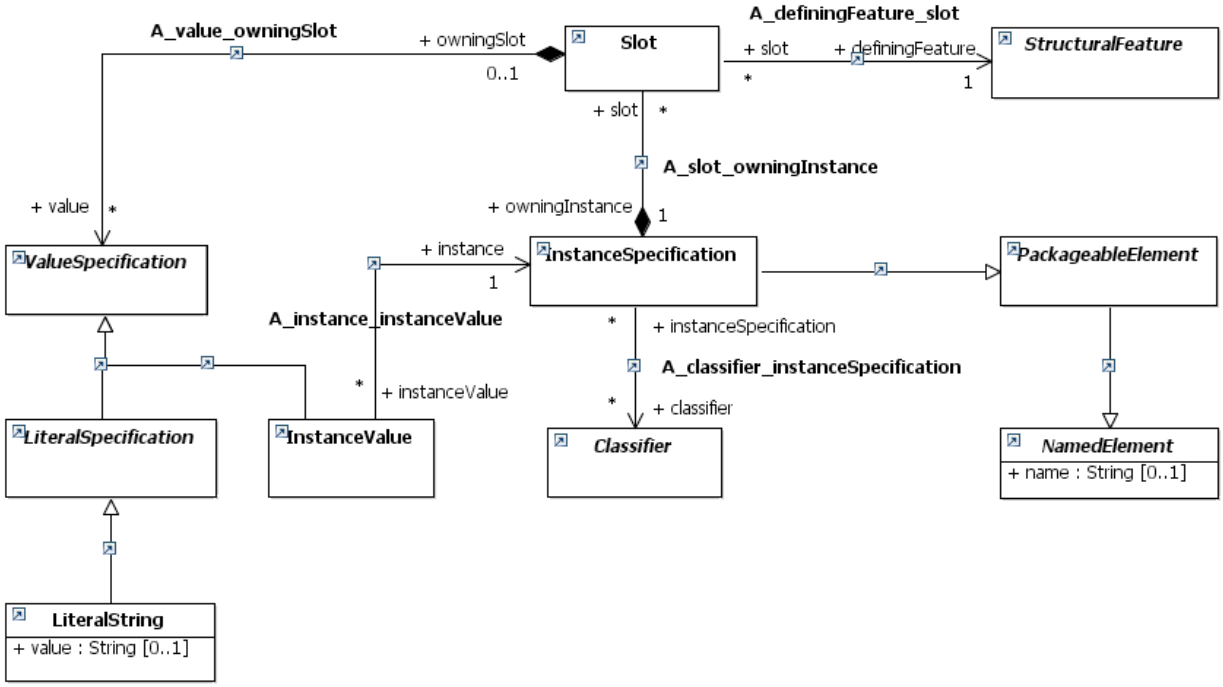

Figure 6-5 Value Specification Abstract Syntax

In order to express the semantics of instance models thus requires additional predicates to relate instance model elements to actual instances at M0. These clearly should be consistent with the predicates `classifies` and `property-value`. For example, suppose we introduce the predicates

(88) `(specifies-instance i x)` means "Instance specification *i* specifies instance *x*."

(89) `(specifies-value v x)` means "Value specification *v* specifies value *x*."

Then we would expect to have semantic relations such as:

(90) "The instance specified by an instance specification is classified by the classifier of the instance specification."

```
(forall (c i x)
    (if (and (A_classifier_instanceSpecification c i)
             (specifies-instance i x))
        (classifies c x)))
```

(91) "The value specified by a slot is the value of the property given by the defining feature of the slot, for the instance specified by the instance specification that owns the slot."

```
(forall (i s v p x y)
    (if (and (A_slot_owningInstance s i)
             (A_value_owningSlot v s)
             (A_definingFeature_slot p s)
             (specifies-instance i x)
             (specifies-value v y))
        (property-value x p y)))
```

(92) "The value specified by an instance value is the same as the instance specified by the instance specification of the instance value."

```
(forall (v i x)
    (if (A_instance_InstanceValue i v)
        (iff (specifies_instance i x) (specifies_value v x))))
```

Of course, by obvious intent, there is a parallelism between the syntactic structure of instance specifications and the property-value structure of the instances being specified. Indeed, an alternate approach to defining the semantics for UML class models would be to specify the semantics of a class model in terms of the set of all possible instance *models* that conform to the class model, rather than in terms of sets of instances form some separate "problem domain."

In this alternate approach, one would, in fact, use syntactic relations such as `A_classifier_instanceSpecification` in the rules for class model semantics, rather than introducing new predicates such as `classifies`. Deduction rules would then, in effect, allow for the direct checking of the consistency of an instance model against a class model and for the derivation of new, valid instance models from instance models already shown to be consistent with the class model. This would be a formal "proof theoretic" approach to specifying class model semantics, entirely at the M1 layer.

This alternate approach is only possible, though, because UML provides an instance modeling syntax rich enough to express all the relations needed to define class model semantics. The approach does not generalize, however, to UML behavior modeling, because UML does not include a notation for expressing the state of an executing behavior. In order to define the semantics of behaviors it is therefore necessary to take the approach of relating the syntactic denotation of behavior (at M1) to a semantic domain of behavior execution (effectively at M0). Since the majority of Clause 10 of the fUML Specification has to do with behavioral semantics, which requires the approach of using a semantic domain, it is only reasonable to use a similar approach for the definition of structural semantics, which is what has been presented here.

6.1.4 Meta-metamodeling

Of course, we are now again left with a given set of predicates derived from UML abstract syntax metamodels such as Figure 6-3 and Figure 6-5. Following the same approach as in Sections 6.1.2 and 6.1.3, we can come up with a meta-metamodel for expressing the UML metamodel. Since the abstract syntax models are all class models, class modeling is all that needs to be covered by this meta-metamodel. Indeed, the OMG Meta Object Facility (MOF) meta-metamodel is based on a subset of the UML infrastructure whose key classes are essentially the same as those shown in Figure 6-3.[10]

Since the semantics of abstract syntax metamodeling is thus exactly that of UML structural semantics, as given by, e.g., the predicates `classifies` and `property-value`, as discussed in Section 6.1.2. Now, the MOF Core Specification actually does provide an "abstract semantics" for Complete MOF (CMOF).[11] This is given in terms of a "semantic domain" that is specified by a class model of instances, similar (though not identical) to the UML instance model abstract syntax given in Figure 6-5. However, this model really just completes the circularity of the MOF specification, because it presupposes that there is already some semantic grounding for the interpreting the instance model itself—which is not provided in any formal way by the UML

[10] See *Meta Object Facility (MOF) Core Specification, Version 2.0,* January 2006, OMG document formal/06-01-01, Subclause 14.1.
[11] *Ibid*, Clause 15.

Infrastructure specification.[12] The formal specification for CMOF can be completed either by using the structural semantics of the CMOF subset of the UML abstract syntax or by providing instance semantics for the CMOF abstract semantic domain model (as outlined at the end of Section 6.1.3 for the UML instance model abstract syntax).

The subset of UML class modeling capabilities used to model the MOF abstract syntax is, essentially, the capabilities included in the MOF meta-metamodel itself. Therefore, it is not advantageous to add any additional meta-layers above that of MOF (i.e., M3). Providing the semantic grounding directly for the structural semantics of MOF then specifies the semantics for the abstract syntax models for all languages defined using MOF.

While MOF does not currently include any behavior modeling capabilities,[13] a similar approach can be taken as for abstract syntax, to define a small subset of UML behavior modeling capabilities that can reflexively model itself. This is essentially what has been done in the definition of the bUML subset in the fUML Specification. It is worth future consideration as to whether a similar subset could be incorporated into MOF to provide a common basis for defining the "abstract semantics" of MOF-based languages, as is now done for abstract syntax.

Until this is done, however, all behavioral semantics must be defined in terms of the specific language metamodel. This is what is done for bUML in Clause 10 of the fUML specification. However, because bUML must currently be a subset of UML superstructure abstract syntax (which is where all the behavioral modeling features are), not the infrastructure, it is not particularly useful to introduce the MOF level at all in its definition. Therefore, the semantic definition for bUML is specified directly in terms of UML M2 abstract syntax predicates, without any consideration of how that abstract syntax is defined in terms of the MOF M3 layer.

6.2 Base Semantics for Time Events

As discussed in Section 4.3.1, the only extension needed to the fUML base semantics (*fUML Specification,* Clause 10) in order to support timing semantics is the semantics of accept event actions triggered by relative time events. The fUML base semantics uses the Process Specification Language (PSL), a foundational axiomatization of processes. The PSL core includes a basic concept of time points, which may be extended with an additional Duration Theory.[14] This can be used to provide an axiomatic basis for the semantics of time events. However, this was not completed within the period of performance of this project.

[12] *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.1.2,* November 2007, OMG document formal/07-11-04.
[13] CMOF does include the ability to model *operations* on metaclasses, but it provides no way to model the *behavior* of these operations. Thus, while it is possible to specify operation behavior using preconditions and postconditions (as used, e.g., in the UML abstract syntax models to specify additional operations for use in well-formedness constraints), it is not possible to operationally define this behavior.
[14] See http://www.mel.nist.gov/psl/psl-ontology/part13/duration.th.html.

Section 7 Reference Implementation

According to the project SOW, the Reference Implementation task is required to:

> *"Develop an open-source reference implementation of the operational semantics specified in [Sections 2-5]. This reference implementation should be able to accept a SysML model in XMI form, per the SysML v1.0 specification, and execute it according to the specified semantics. This implementation should extend and encompass any reference implementation developed to support the Foundational UML submission in [Section 1]."*

During the period of performance of this project, a reference implementation was completed for the fUML subset defined in the *fUML Specification,* including an implementation of the Foundation Model Library (except that the Basic Input/Output implementation was limited to the StandardOutputChannel). However, no SysML capabilities beyond fUML were implemented.

7.1 Source Code

The Reference Implementation is written in Java. Per the SOW, the code written for this project is intended to be licensed as open source by Lockheed Martin. Other than the new code developed on the project, the reference implementation uses the libraries listed in Table 7-1, which are all also licensed as open source. Further detail on the Reference Implementation source code (other than that provisioned directly from models) can be found in the Reference Implementation *Software Design Description,* which is attached to this report.

Table 7-1 Third Party Libraries

Name	Vendor	Version	License
Sun Java Streaming XML Parser (SJSXP)	Sun Microsystems	1.0.1	CDDL v1.0 / GPL v2[a]
Java Architecture for XML Binding (JAXB)	Sun Microsystems	2.1 EA2	CDDL v1.0 / GPL v2[a]
log4j	Apache Software Foundation	1.2.8	Apache Software License v2.0[b]
JUnit	junit.sourceforge.net		CPL v1.0[c]

[a] See https://glassfish.dev.java.net/public/CDDL+GPL.html
[b] See http://logging.apache.org/log4j/1.2/license.html
[c] See http://junit.sourceforge.net/cpl-v10.html

7.2 Execution Engine

The core of the Reference Implementation is an execution engine for fUML models. As shown schematically in Figure 7-1, the source code for the execution engine is generated (or *provisioned*) from the semantic execution model defined in the *fUML Specification.* This provisioning was performed using the Model Driven Solutions ModelPro tool (which is to made available under an open source license in early 2009).

The provisioned execution engine source code is combined with hand-written code for loading XMI, to produce the executable Java archive file. When the execution engine is run, it configures itself for loading fUML models based on the fUML abstract syntax model, as given in the *fUML Specification.*

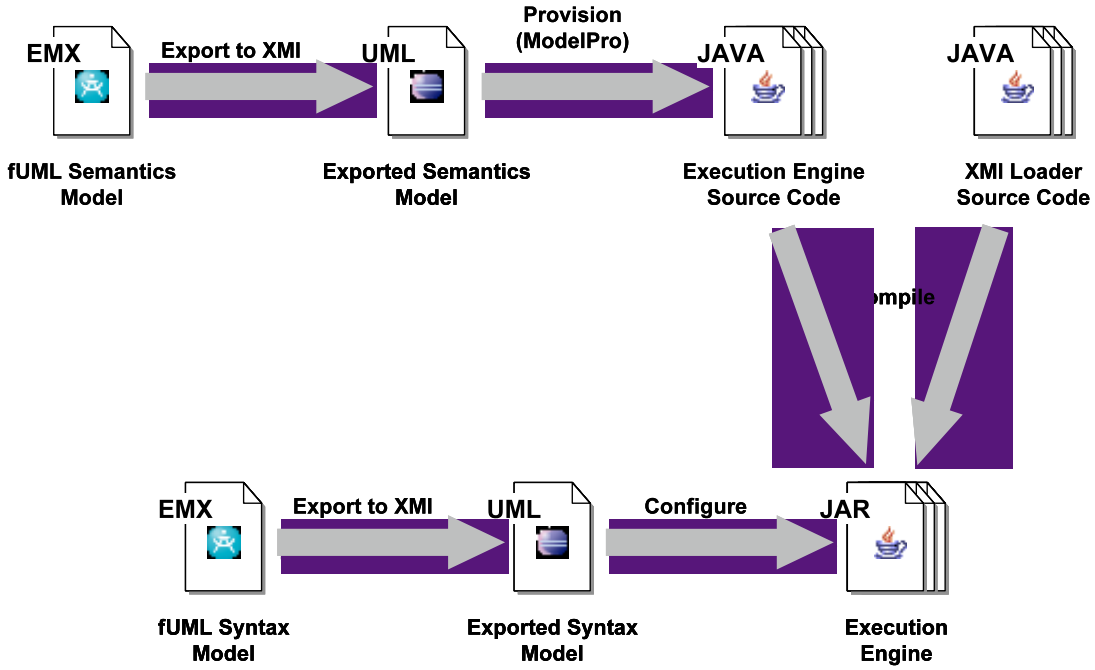

Figure 7-1 fUML Execution Engine

Once the execution engine has configured itself, it can then load a user model in XMI format (see Figure 7-2). Given the name of a top-level activity in the model, the execution engine uses that activity as the root to begin execution. A textual execution trace is written to the standard system output.

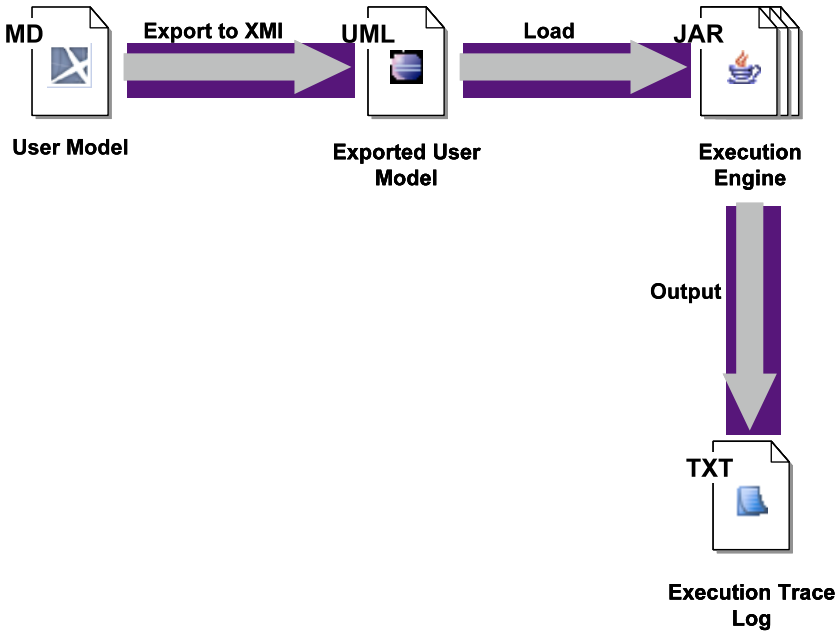

Figure 7-2 User Model Loading

7.3 Foundation Model Library Implementation

The Reference Implementation includes an implementation of the Foundation Model Library, as specified in Clause 9 of the *fUML Specification.* This includes a complete implementation of all Primitive Behaviors (Subclause 9.2) and a partial implementation of Basic Input/Output (Subclause 9.4). From Basic Input/Output, only the StandardOutputChannel class and the WriteLine behavior are implemented.

The integration of the Foundation Model Library implementation is challenging, because a user model references standard library model elements, but execution engine then needs to execute these using implementation code specific to the Reference Implementation (see Figure 7-3). The needed linkages are made using a configuration table that associates the Java implementation class with the normative XMI ID of the library element being implemented.

When the Foundation Model Library is loaded into memory, the XMI IDs for primitive behaviors are resolved to the corresponding abstract syntax objects in memory. These are then used to register implementation objects for each behavior with the execution factory. Then, when a user model is loaded, XMI references to primitive behavior library model elements are similarly resolved to the appropriate abstract syntax element in memory. When such a primitive behavior is executed, the execution factory can then produce the correct implementation for it. (For more on the configuration of the fUML execution environment, see Subclause 8.2.1 of the *fUML Specification.*)

In addition to the registration of the primitive behaviors, a single instance of StandardOutputChannel is created at the locus of the execution environment. This instance is implemented as a special library instance that dispatches operation calls to library implementation code. A user model can then obtain the standard output channel instance using a read extent action and call operations on it, or use the standard WriteLine convenience behavior (see Subclause 9.4 of the *fUML Specification*).

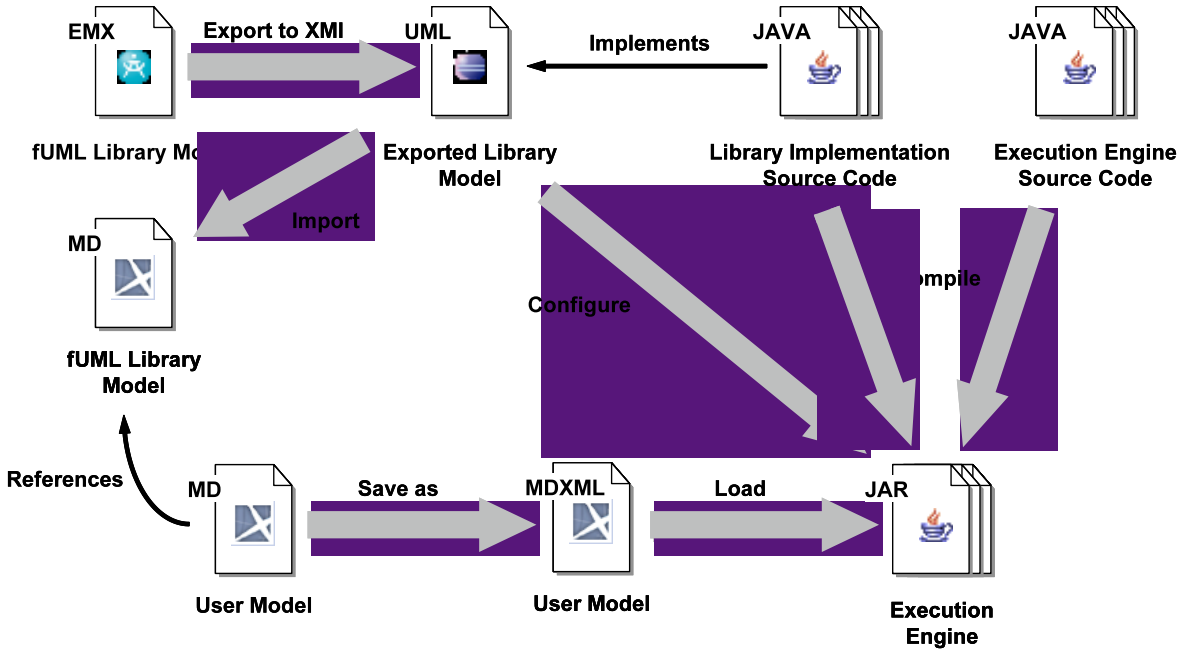

Figure 7-3 Foundation Model Library Integration

Section 8 Proposed Changes to UML and SysML

According to the project SOW, the Proposed Specification Changes task is required to:

> *"Identify any required changes to the UML and/or SysML specifications need to support the above [tasks]."*

8.1 Changes Based on the fUML Specification

The intent of the *Semantics of a Foundational Subset for Executable UML Models* submission was to provide a precise semantics for the foundational subset (fUML) that was consistent with the less formal semantic specification of the current UML 2 Superstructure Specification. This, of course, required some interpretation of the semantic descriptions in the Superstructure Specification. In just a few cases, however, the interpretation for fUML must be considered to not be entirely consistent with the current specification.

Subclause 6.1 of the submission identifies the following cases of inconsistency (the cross references are to subclauses of the submission document).

- Null tokens are not passed to the output pins of an invoking action when the corresponding parameter nodes of an invoked activity are empty and the activity is terminated (see Subclause 8.5.2).
- Rather than receiving a full collection on a single token, expansion nodes use the set of all tokens they are holding as the "collection" referenced by their expansion region (see Subclause 8.5.4).
- Test identity actions apply to inputs that are data values, as well as objects, testing data values for equality by value rather than identity of reference (see Subclause 8.6.3).

The submission specifically states that the fUML specification does *not* change the current UML specification to resolve these consistencies. Instead, the UML 2.3 RTF process will be used to recommend proposed changes to the UML specification to make the general semantics consistent in these areas with the detailed semantics chosen for fUML.

8.2 Changes Based on This Project

As a result of work on the detailed specification of the semantics of streaming given in Section 3.1, a single change to UML 2 has been identified to be proposed to the UML 2.3 RTF (see also the discussion under Reentrancy in Section 3.1.2.1).

- The description of the Behavior::isReentrant attribute in Subclause 13.3.2 is "Tells whether the behavior can be invoked while it is still executing from a previous invocation." Further, the semantics section for Action (Subclause 12.3.2) states "If a behavior is not reentrant, then no more than one execution of it will exist at any given time." Together, these seem to imply that it is only possible to have one execution at all *globally* for a non-reentrant behavior at any one time. This seems to be a stronger requirement than necessary for non-reentrant behaviors—and it is certainly stronger than necessary for streaming. A less restrictive but sufficient requirement would be that there can only be at most one execution of a non-reentrant behavior at the same time *invoked from the same call action.* This would allow two different call actions to invoke the same non-reentrant behavior concurrently.

In addition, work on the semantics for call actions has resulted in the identification of a semantic error in the *fUML Specification* that will be proposed for correction by the fUML FTF (see also the note at the end of Section 3.1.2.5).

- The callExecution attribute of the CallActionActivation class currently has a multiplicity of 0..1 (see Subclause 8.6.2 of the *fUML Specification*). However, since all behaviors are required to be reentrant in fUML, it is allowable for a single call action to fire multiple times concurrently. But, in such a case, there will be more than one ongoing execution invoked from the call action. Therefore, the callExecution attribute should be replaced with a callExecutions attribute having multiplicity *, and the operations specified for CallActionActivation should be updated to reflect this.

Further, work on the semantics of timing and profiles suggests the need for consideration of the following fairly significant updates to the design of the fUML execution model.

- A more explicit model of threading within the execution model would both make the concurrent activity execution semantics of fUML clearer and provide the hooks required for thread suspension and resumption required to model the semantics of timing (see Section 4.3.3).
- Using getter and setter operations for all access to the attributes of classes in the execution model would make it possible to model the semantics of applied stereotypes using a semantic wrapper approach (see Section 5.1.2).

These updates will be proposed to the fUML FTF as potential changes to the *fUML Specification* to be made before finalization.

No necessary changes to SysML were identified during the period of performance of this project.