An OMG® Executable UML® Publication

OBJECT MANAGEMENT GROUP

Semantics of a Foundational Subset for Executable UML Models (fUML)

Version 1.5 – Beta

OMG Document Number: ptc/2020-05-10

Date: May 2020

Normative reference: https://www.omg.org/spec/FUML/1.5

Machine readable file(s):

Normative: https://www.omg.org/spec/FUML/20200501/fUML_Syntax.xmi
https://www.omg.org/spec/FUML/20200501/fUML_Semantics.xmi
https://www.omg.org/spec/FUML/20200501/fUML_Library.xmi

Table of Contents

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. All OMG specifications are available from the OMG website at:

http://www.omg.org/spec

OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the link cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Issues

The reader is encouraged to report any technical or editing issues/problems with this by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under documents, Report a Bug/Issue.

1 Scope

The scope of this specification is the selection of a subset of the UML 2 metamodel that provides a shared foundation for higher-level UML modeling concepts, as well as the precise definition of the execution semantics of that subset. Given its fundamental nature, the subset assumes the most general type of system, including physically distributed and concurrent systems with no assumptions about global synchronization.

Many executable UMLs are conceivable, based on executing use cases, activities, workflow, methods, or state machines and their combinations. This specification covers the capabilities shown in the Structural Modeling layer of Figure 6.1, subclause 6.3.2 of the UML 2Specification, as well as the Common Behavior, Actions and Activities capabilities in the Behavioral Modeling layer. This functionality is covered primarily in Clauses 7-12 and Clauses 15-16 of the UML 2 Specification.

The selected elements are translatable into an implementation such that a specified functional computation is independent of the control and data structures in which the elements reside. This translatability provides maximum flexibility to modify the organization of the data without affecting the definition of an algorithm. (The UML 1.5 action metamodel was designed in this manner for precisely this reason.)

It is not the intent of this specification to define the specification of every higher-level UML construct in terms of elements from the foundational subset; however, the specification does intend to encourage use of the broadest possible subset of UML constructs that can be reduced to a small set of elements.

In sum, the foundational subset defines a basic virtual machine for the Unified Modeling Language, and the specific abstractions supported thereon, enabling compliant models to be transformed into various executable forms for verification, integration, and deployment.

2 Conformance

2.1 General

This specification defines a subset of UML 2 and specifies foundational execution semantics for it. This subset will be referred to as Foundational UML or "fUML." Conformance to this specification has two aspects:

- Syntactic Conformance: A conforming model must be restricted to the abstract syntax subset defined for fUML.
- Semantic Conformance: A conforming execution tool must provide execution semantics for a conforming model consistent with the semantics specified for fUML.

The fUML syntactic subset is defined by the abstract syntax subset model given in Clause 7. The packages in this metamodel correspond to similarly named packages in the UML 2 metamodel, which act as the basic language units for the purpose of syntactic conformance. The semantics for fUML is specified by the execution model given in Clause 8. The packaging structure of the execution model parallels the language unit packaging of the fUML abstract syntax exactly, except that there are no semantics packages for "Common Structure" or "Packages", and there is one additional package called "Loci."

2.2 Meaning and Types of Conformance

Conformance to a specific fUML conformance level entails both syntactic *and* semantic conformance. Syntactic conformance is defined in terms of a *conforming model*.

- *Abstract Syntax Conformance* – A UML model *conforms* to fUML if it is a well-formed model constructed from only syntactic elements that are included in the fUML abstract syntax subset. A *well-formed* model is one that meets all constraints imposed on its syntactic elements by the UML 2 abstract syntax metamodel *and* any additional constraints imposed on those elements in the fUML subset (given in Clause 7).

- *Model Library Conformance* – In addition, a conforming fUML model may make use of elements from the fUML model library (see Clause 9). An execution tool is not required to implement any of the model elements defined in Clause 9, but, if such elements are provided, they must conform to the behavior specified in that clause. An execution tool may, in addition, make available a tool-specific model library for use by conforming models accepted by the tool, so long as the execution behavior of elements of the models in that library may be entirely defined in fUML.

The fUML specification provides a precise definition of the execution semantics for a conforming model. Conformance to these semantics is defined in terms of a *conforming execution tool* (see Clause 4 for the definition of the term "execution tool" as used in this document). If a conforming execution tool is presented with a conforming model, then it must behave as further described below. On the other hand, if it is presented with a non-conforming model, then it may react in one of the following three ways.

- *Rejection* – It may reject the model and refuse to process it further at all.
- *Static Partial Acceptance* – If the tool is able to statically determine that the non-conforming parts of the model are all elements of abstract syntax packages that are not included in the fUML subset at all, and that the model elements from packages included in the fUML subset all conform to fUML, then the tool may accept the model. In this case, any elements that are not included in the fUML subset, and are not instances of metaclasses that are specializations, directly or indirectly, of metaclasses in the fUML subset, may be ignored by the tool. Any elements that are not included in the fUML subset, but are instances of metaclasses that are specializations of metaclasses in the fUML subset, must be interpreted as if they are instances of the superclass that is in the fUML abstract syntax.
- *Dynamic Partial Acceptance* – The tool may accept the model for execution and attempt to evaluate or execute any value specification or behavior from the model, interpreting any model elements as in the case of static partial acceptance. However, if the tool encounters any model element that is defined in an abstract syntax within the fUML subset, but does not conform to the additional constraints defined for the fUML subset, then the tool must terminate execution with an error.

A conforming execution tool need not use the same option above in all cases. However, it must be specified for any conforming tool in which cases each option is used.

To further claim conformance for an execution tool at a specific level, it must be possible to demonstrate the following:

- *Abstract Syntax Mapping* – An execution tool accepts a UML model for execution in some concrete form. It must be possible to bidirectionally map this concrete input form to a well-formed representation in terms of instances of the metaclasses in the fUML abstract syntax at the given conformance level. One standard way to do this is to use the XML Metadata Interchange (XMI) as the input form for the model, in which case the mapping to the UML abstract syntax is provided by the XMI standard (see Clause 3). However, it is not required that XMI be used as the input form. For example, a tool may provide for direct model input in terms of graphical and or textual notation, so long as this may be fully mapped to the fUML abstract syntax.
- *Semantic Value Mapping* – Runtime inputs and outputs are semantically specified by a model of *values* (see 8.4 to 8.7). During the execution of a behavioral model, the model execution will generally take values as inputs and produce values as outputs. The execution tool must provide a concrete implementation for all such values and demonstrate a mapping from this implementation to the model of values provided in the execution model. For this mapping, it is only required to demonstrate the effective implementation of the *properties* defined for the value classes, showing the corresponding implementation value for any value instance from the semantic model, and vice versa. It is *not* required to demonstrate the implementation of the operations specified for those classes in the execution model. Also, if the execution tool uses different internal and external forms for values, it is only required to provide a mapping for the external form, so long as this is sufficient to demonstrate semantic conformance, as described below.

- *Execution Environment Mapping* – The fUML execution model provides an abstraction of the *execution environment* for a model in terms of the concept of an execution *locus* (see 8.3). It must be possible to demonstrate how the actual execution environment provided by an execution tool corresponds to the locus concept. Specifically, this must include:
  - A definition of whether execution takes place at a single locus or may be distributed across multiple loci. If the latter, then the tool must provide a mechanism for allocating a model or a portion of a model to a specific locus.
  - A description of whether and how extensional values (see 8.3 and 8.7) are persisted at a locus across behavior executions.
  - A specification of what objects are pre-instantiated at a locus in order to provide system services (such as input/output—see 9.4).

Note that, for an execution tool that, say, compiles a model to some target executable form, the execution environment for the purposes of this mapping will be the environment in which the target executable runs, rather than the environment of the tool itself.

- *Semantic Conformance* – Finally, a conforming execution tool must provide an implementation of the interface of the Executor class from the execution model (see 8.3). While it is not necessary that this be a strict implementation of the object-oriented operations provided by Executor, it must be possible to demonstrate the following:
  - *Evaluation* – Given a well-formed value specification from a conforming model, the tool must be able to produce a value conforming to the result of the Executor::evaluate operation on the value specification.
  - *Synchronous Execution* – Given a well-formed behavior from a conforming model and values for all input parameters of the behavior, the tool must be able to execute the behavior in conformance to the effect and results of the Executor::execute operation.
  - *Asynchronous Execution* – Given a behavior or an active class from a conforming model, the tool must be able to asynchronously start the given behavior in conformance to the effect and results of the Executor::start operation.

Note that, at a given conformance level, a conforming execution tool must semantically conform when presented with *any* conforming model at that level. That is, to conform at a certain level, an execution tool must implement *all* of the fUML abstract syntax at that level and provide conforming semantics for it.

The above definition of semantic conformance uses the concept of *conforming to an operation* of the Executor class from the execution model. This concept is further defined as follows:

- Inputs provided to the execution tool must correspond to the input parameters required for the operation.
- Using the abstract syntax and semantic value mappings for the tool, map the inputs to the execution tool from their implementation form to the corresponding representation in terms of instances of abstract syntax and semantic value classes.
- Using the execution environment mapping, map the intended target execution environment to a corresponding model in terms of execution loci and pre-instantiated extensional values.
- Using the specification of the given operation as part of the execution model (or the subset of that model that applies at a certain conformance level), determine the effect of invoking the operation on the given input values using an executor at a specific execution locus. This includes the generation of output values and any side effects that occur at and through the execution locus.
- Using the execution environment mapping, map any updates to loci to updates to the target execution environment.
- Using the semantic value mapping, map any output values to the corresponding implementation form for the tool.
- Conformance requires that the actual outputs and environmental changes produced by the execution tool be *consistent* with the outputs and changes determined in the two bullets directly above.

The conformance requirement here is one of *consistency* rather than *equivalence* because, as a semantic specification, the execution model tightly *constrains*, but does not always fully determine, the exact results of an execution. This is particularly true in the presence of the high degree of concurrency possible with UML activity models, in which different conforming implementations may produce significantly different resulting executions of the same model due to timing issues.

This allowance for some flexibility in the conformance requirements is known as the *genericity* of the execution model, (discussed in more detail in 2.3). Nevertheless, it is still possible to formalize the conformance requirements even in the presence of such genericity.

- Clause 10 specifies the *base semantics* for the execution model. This specification effectively provides for an interpretation of the execution model as a set of first-order predicates, or *axioms,* over possible execution traces.
- A specific invocation of an operation in the execution model, as called for in the determination of conformance to the operation above, results in an execution trace. Any execution trace that satisfies the axioms of the base semantics is a legal execution trace.
- Conformance to the operation requires that the execution tool conform to the effect and results of *any* legal execution trace of the operation. The tool is allowed to conform to different execution traces for different invocations of the operation, even on identical inputs in an identical environment.

In essence, the base semantics provides an interpretation of the execution model as a set of constraints on the allowable execution of well-formed fUML models. A conforming execution tool must produce results that do not violate these constraints, but there is flexibility for allowing different implementations to provide somewhat different behavior for the execution of the same well-formed model, within the specified constraints. Ideally, conformance would be demonstrated by a formal proof that the execution tool implementation meets all the required constraints. In reality, it is expected that conformance will be demonstrated by a sufficient suite of tests hand checked against the specification, as is the case for, say, conformance to most major programming language standards.

2.3 Genericity of the Execution Model

To support a variety of different execution paradigms and environments—including a number of widely used commercial and research variants of executable UML —the specification of the execution model incorporates a degree of *genericity*. This is achieved in two ways: (1) by leaving some key semantic elements unconstrained, and (2) by defining explicit semantic variation points. A particular execution tool can then realize specific semantics by suitably constraining the unconstrained semantic aspects and providing specifications for any desired variation at semantic variation points.

The semantic areas below are not explicitly 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.
- *The semantics of concurrency* – The execution model includes an implicit concept of concurrent threading of execution (see the discussion in 8.9.1). However, it does not require that a conforming execution tool actually execute such concurrent threads in a physically parallel fashion and it is agnostic about the actual scheduling of execution of concurrent threads that are not physically executed in parallel. So long as the execution tool respects the various creation, termination, and synchronization constraints placed on such threads by the execution model, any sequentially ordered, or partial or totally parallel, execution of concurrent threads conforms to a legal execution trace.
- *The semantics of inter-object communications mechanisms* – This refers specifically to communication properties of the medium through which signals and messages are passed between objects. The execution model is written as if all communications were perfectly reliable and deterministic. However, this is not realistic for all execution tool

implementations. Therefore, despite the restrictions that would be imposed by a strict interpretation of the execution model, conformance of an execution tool to the semantics of inter-object communication is not predicated on any assumptions about whether or not such communication is reliable (i.e., that signals and messages are never lost or duplicated), preserves ordering, happens with deterministic or non-deterministic delays, and so on.

Different execution tools may semantically vary in the above areas in executing the same model, while still being conformant to the semantics specified by the execution model for fUML. Additional semantic specifications or constraints may be provided for a specific execution tool in these areas, so long as it remains, overall, conformant to the execution model. For instance, a particular tool may be limited to a single centralized time source such that all time measurements can be fully ordered.

In contrast to the above areas, the items below are explicit *semantic variation points.* That is, the execution model as given in this specification by default fully specifies the semantics of these items. However, it is allowable for a conforming execution tool to define alternate semantics for them, so long as this alternative is fully specified as part of the conformance statement for the tool.

- *Event dispatch scheduling* – As described in 8.8, event occurrences received by an active object are placed into an *event pool.* The event occurrences in the pool are then asynchronously dispatched, potentially triggering waiting accepters of such events. By default, events are dispatched from the pool using a first-in first-out (FIFO) rule. However, a conforming execution tool may define an alternative rule for how this dispatching is scheduled by providing a specialization of the GetNextEventStrategy class that redefines the *dispatchNextEvent* operation to specify the desired rule.

- *Polymorphic operation dispatching* – Operations in UML are potentially polymorphic—that is, there may be multiple methods for any one operation. The determination of which method to use for a given invocation of the operation depends on the context and target of the invocation. The specification for this determination is provided in the execution model by the *dispatch* operation of the Object class, as specified in 8.7 (the semantics of operation dispatching is further discussed in relation to the call operation action in 8.9). By default, the method used for an operation must be associated with a (possibly inherited) member operation of a type of the target object of the operation invocation that is either the invoked operation or a redefinition (“override”) of it. However, a conforming execution tool may define an alternative rule for how this dispatching is to take place by providing a specialization of the DispatchStrategy class that redefines the *dispatch* operation to specify the desired rule.

If a conforming execution tool wishes to implement a semantic variation in one of the above areas, then a specification must be provided for this variation via a specialization of the appropriate execution model class as identified above. This specification must be provided as a fUML model in the “base UML” subset interpretable by the base semantics of Clause 10. Further, it must be defined in what cases the variation is used and, if different variants may be used in different cases, when each variant applies and/or how what variant to use is to be specified in a conforming model accepted by the execution tool.

2.4 Conformance Statement

The conformance of an execution tool to the fUML specification may be summarized in a *conformance statement* for the tool. Such a statement should include the following items.

- *Model Library* – An identification of what elements of the standard fUML model library are implemented by the tool. A specification of any additional tool-specific model library elements.

- *Abstract Syntax Mapping*

- *Semantic Value Mapping*

- *Execution Environment Mapping*

- *Semantic Conformance* – A demonstration of semantic conformance in terms of the above mappings.

- *Semantic Constraints* – A specification of any additional semantic constraints on semantic areas left unconstrained by the execution model.
- *Semantic Variation* – For each semantic variation point, a specification of any variation from the default semantics.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification.

The following OMG standards provided the source for the foundational subset.

- UML 2.5.1 Specification, http://www.omg.org/spec/UML/2.5.1
- MOF 2.5.1 Core Specification, http://www.omg.org/spec/MOF/2.5.1
- OCL 2.4 Specification, http://www.omg.org/spec/OCL

XML Metadata Interchange (XMI) provides a syntactic interchange mechanism for models. It is expected that models conforming to this specification will be interchanged using XMI.

- MOF 2.5.1 XMI Mapping Specification, http://www.omg.org/spec/XMI/2.5.1

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Base Semantics

A definition of the execution semantics of those UML constructs used in the execution model, using some formalism other than the execution model itself. Since the execution model is a UML model, the base semantics are necessary in order to provide non-circular grounding for the execution semantics defined by the execution model. The base semantics provide the "meaning" for the execution of just those UML constructs used in the execution model. The execution model then defines the "meaning" of executing any UML model based on the full foundational subset. Any execution tool that executes the execution model should reproduce the execution behavior specified for it by the base semantics.

Behavioral Semantics

The denotational mapping of appropriate language elements to a specification of a dynamic behavior resulting in changes over time to instances in the semantic domain about which the language is making statements.

Compact Subset

For the purposes of this specification, a compact subset of UML is one that includes as small a subset of UML concepts as is practicable to achieve computational completeness.

Computationally Complete

A computationally complete subset of UML is one that is sufficiently expressive to allow definition of models that can be automatically executed on a computer by an execution tool.

Execution Model

A model that provides a complete, abstract specification to which a valid execution tool must conform. Such a model defines the required behavior of a valid execution tool in carrying out its function of executing a UML model and therefore provides a definition of the semantics of such execution.

Execution Semantics

For the purposes of this specification, the behavioral semantics of UML constructs that specify operational action over time, describing or constraining allowable behavior in the domain being modeled.

Execution Tool

Any tool that is capable of executing any valid UML model that is based on the foundational subset and expressed as an instantiation of the UML 2 abstract syntax metamodel. This may involve direct interpretation of UML models and/or generation of equivalent computer programs from the models through some kind of automated transformations. Such a tool may also itself be concurrent and distributed.

Foundational Subset

The subset of UML to which execution semantics are given in order to provide a foundation for ultimately defining the execution semantics of the rest of UML.

Static Semantics

Possible context sensitive constraints that statements of a language must satisfy, beyond their base syntax, in order to be well-formed.

Structural Semantics

The denotational mapping of appropriate language elements to instances in the semantic domain about which the language makes statements.

Syntax

The rules for how to construct well-formed statements in a language or, equivalently, for validating that a proposed statement is actually well-formed.

5 Symbols

There are no symbols or abbreviated terms necessary for the understanding of this specification.

This page intentionally left blank

6 Additional Information

6.1 Changes to Adopted OMG Specifications

The Foundational Subset for Executable UML Models specification does not change any adopted OMG specifications. The semantics defined in this specification are generally a precise definition of a subset of the UML semantics given in the UML 2 Specification. For this subset, the foundational execution semantics are intended to be consistent with, though sometimes more restrictive than, the less precise textual semantic specification given in the UML 2 Specification. Cases where the foundational execution semantics restrict some semantic variability allowed in the UML 2 Specification are noted in the overview discussions in the subclauses of Clause 8, Execution Model.

6.2 On the Semantics of Languages and Models

In a general sense, a *language* is a symbolic means for communication. The language provides rules for constructing *statements* that communicate some specific meaning. In a natural language, these rules evolve neurologically and socially over time. For a formal language, on the other hand, the rules are constructed artificially in order to create a means of communication that, for some intended purpose, is in some way more precise than natural language.

A formal language only attaches meaning to statements that are correctly constructed or *well formed*. The *syntax* of the language provides the rules for how to construct well-formed statements or, equivalently, for validating that a proposed statement is actually well-formed. The *semantics* of the language then provides the specification of the meaning of well-formed statements.

It is usually possible to completely specify the syntax of a formal language. This is because syntax has specifically to do with the form and structure of statements in the language. Semantics is more problematical because it is inherently extrinsic to the form of the statements themselves. Meaning can only be assigned to a formal statement in relation to entities in some *semantic domain* about which the statement is intended to communicate.

An *interpretation* of a statement is a mapping of syntactic elements of the language to elements of the semantic domain such that the truth-value of the statement can be determined, to some level of accuracy. Colloquially, an interpretation of a model can be said to give it "meaning" relative to the semantic domain. If this mapping can be inverted, so that elements of the semantic domain can be mapped to syntactic language elements, then a statement can also be constructed as a *representation* of some part of the semantic domain, such that the statement is true under the interpretation mapping.

As a somewhat stylized example from natural language, consider the simple statement "Jack owns that house." This is a syntactically correct statement in the English language. We can interpret the statement in terms of the "real world" as the semantic domain.

The word "Jack" is a syntactic element that *denotes* some person in the real world under this interpretation. Similarly, the phrase "that house" denotes a specific structure in the real world. Finally, the word "owns" denotes a legal relationship that may hold between a person and property. If this legal relationship does exist between the previously identified person and structure, then we can say that the statement "Jack owns that house" is true under this interpretation. Otherwise it is false.

Conversely, suppose we know it to be true that a person named "Jack" has legal ownership of a specific house being pointed to. Then we can say that the statement "Jack owns that house" is a truthful representation of this situation.

One of the most useful aspects of a formal language is that it can be used to make concrete statements about potentially abstract elements of the semantic domain. Essentially syntactic manipulations of these statements can then be used to make deductions about the semantic elements represented by the statements.

A *theory* is structured set of rules for *deducing* new statements in a language from existing statements. A theory is considered *correct* under a certain interpretation if any statements deduced from true statements under the interpretation are themselves always true. In this way, the syntactic deduction rules of the theory may be used to make corresponding actual deductions in the semantic domain.

A *model* is a set of statements in a *modeling language* about some domain under study, which provides the semantic domain for the model. The meaning of statements in the model is then assigned by an interpretation that maps model elements to elements of that semantic domain.

A model may be used to *describe* a domain. In this case, the model is considered *correct* (under some interpretation) if all statements made in the model are true for the domain. Similarly, a theory is considered correct for this domain if all statements deduced using the theory from statements in the model are also true.

Alternatively, a model may be used as a *specification* for a domain (or for some system within a domain). In this case, a specific domain is considered *valid* relative to this specification if no statement in the model is false for the domain. Similarly, a valid domain *conforms* to a specific theory if, in addition, no statements deducible using the theory from the model are false. That is, all statements deducible from the model also effectively become part of the specification.

UML is, of course, a modeling language. "Statements" in UML are constructed using a combination of (syntactic) modeling elements, both graphical and textual. The statements made by a UML model can then be interpreted against the domain being modeled.

For example, consider the simple instance model shown in Figure 6.1. As a model of the "real world," this can be interpreted as making the set of statements: "There is a person whose name is Jack. There is a house. The person is the owner of the house."

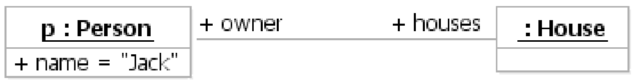


Figure 6.1 - Simple UML Instance Model

Note that it is an *instance* model that is interpreted here as making direct statements about the real world. These statements are what logicians call *first order* propositions. However, it is more common in UML to model (at least initially) at the level of classes. A class model makes *second order* statements about what kind of first order propositions are valid for the domain under study.

Consider the class model in Figure 6.2. Structurally, this model *requires* that each instance of the class Person have the properties "name" and "houses." Further, it *requires* that the name of an instance of Person have a String value and it *allows* the instance to have zero or more houses associated with it.

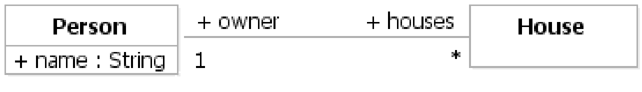


Figure 6.2 - Simple UML Class Model

Under this interpretation, the relationship between the instance model of Figure 6.1 and the class model of Figure 6.2 is basically one of consistency. The instance p in Figure 6.1 is declared to have the class Person as its type. It is therefore required to have a *name* attribute. It would certainly be possible to construct a UML model of an instance of Person that does not have a name. However, this would be *inconsistent* with the class model given in Figure 6.2.

Of course, it is also possible to give a direct interpretation of a class model in terms of the domain under study. For example, we could take the class Person to denote the set of all people and the class House to denote the set of all houses, while the association Ownership denotes a relationship between people and houses. The class model of Figure 6.2 then makes statements about the "real world" such as "Every person has a name" and "Some people own houses" (where the latter statement reflects the "zero or more" multiplicity of the "houses" association end).

There is also another common, but very different, interpretation that may be given to the same class model shown in Figure 6.2. In this interpretation the domain under study is that of computer programs written, say, in the Java programming language. That is, the class model is interpreted as a model *of* a Java program in this domain. Each class in the model is taken to denote a corresponding Java class with each property in the model denoting a corresponding field in the Java class. If the class model is taken as a specification, then the model will actually exist before the program is written; the model becomes the design for constructing a valid Java program.

This example points out the fact that the same model may have different "meanings" under different interpretations. In fact, it may even be useful to have multiple interpretations for a model at the same time. Indeed, under the usual tenets of object-oriented design, the design model of program should *also* be interpretable as a model (in a somewhat restricted sense) of the portion of the real world relevant to the program (the so-called "problem domain").

6.3 On the Semantics of Metamodels

A *metamodel* is often rather loosely defined as "a model of a model." For our purposes here, however, a more precise definition is "a model of a modeling *language*." Thus, the UML metamodel is a model with the domain under study being UML, the language.

Another way to look at this is to consider the metamodel to be a specification model for a *class* of "systems" in the semantic domain, where each system in the class is itself a valid model expressed in a certain modeling language. The metamodel therefore makes statements about what can be expressed in the valid models of the modeling language. Since a metamodel is a specification, a model in the modeling language is *valid* only if none of these statements are false.

If the interpretation mapping for a metamodel is invertible, one can also uniquely map elements of the modeling language back to elements of the metamodeling language. In this case, given any model, we can invert the interpretation mapping to create a *metamodel representation* of the model; that is, a set of true statements about the model expressed in the metamodeling language.

A *theory of a metamodel* is a way to deduce new statements about a modeling language from the statements already in a metamodel of the modeling language. Since a metamodel is a specification, a valid model in the modeling language must not violate any statement deducible using the theory from the explicit metamodel statements.

One way to look at this is to consider the statements of the metamodel as *axioms* about the modeling language. Then, given the metamodel representation of a model, we can *deduce*, using the theory, whether the representation of the model is *consistent* with the metamodel. If it is consistent, then the model is valid, otherwise it is not.

The UML Specification provides a metamodel of UML. That is, it includes a set of statements about UML models that must not be violated by any valid UML model. Note that, in its entirety, this metamodel can be considered to include all of the concrete graphical notation, abstract syntax and semantics for UML. However, as defined in the Specification, the only part of this metamodel that is *formal* is the abstract syntax model.

The UML abstract syntax is formalized as a UML class model. It is thus an example of a *reflexive* metamodel. That is, it is expressed in the same modeling language that it is defining. This, of course, introduces an inherent circularity.

Since a reflexive metamodel is expressed in the same modeling language as it is describing, its interpretation provides a mapping of the modeling language onto itself. Generally, this mapping will be from the entire modeling language to a subset of it. One can then iterate this mapping, each time producing a smaller subset, until one reaches the *minimal* reflexive metamodel that maps completely onto itself, rather than a subset. This minimal metamodel contains the smallest set of modeling elements required in order to specify the modeling language in question.

An interpretation of a *minimal* reflexive metamodel maps the metamodel onto itself. This means that any statement in the minimal reflexive metamodel can be represented in terms of elements of the minimal reflexive metamodel. However, the interpretation of this representation is itself expressed reflexively as a mapping to yet another representation in terms of the minimal reflexive metamodel. This circularity means that, for a minimal reflexive metamodel, the interpretation mapping really provides no useful expression of the “meaning” of the metamodel itself. To break this circularity, the minimal reflexive metamodel must be given a *base* semantics that is independent of its circular interpretation in terms of itself.

In the case of UML, the “minimal” reflexive abstract syntax metamodel is the UML Infrastructure (for pragmatic reasons the Infrastructure is not actually absolutely “minimal,” but it is still just a small subset of the full UML Superstructure). The Meta-Object Facility (MOF) specification defines a standard meta-metamodel based on the UML Infrastructure that provides the basic elements required to construct the abstract syntax metamodel for any modeling language.

The MOF specification also attempts to provide an “Abstract Semantics” for the MOF meta-metamodel. However, this semantics is still defined in terms of a semantic domain that is specified using a UML class model. Thus, the circularity is not really broken. The only interpretations of the MOF meta-metamodel that are effectively non-circular are those provided by the standard mappings of the meta-metamodel to other technologies, such as XML Metadata Interchange (XMI) and Java Metadata Interface (JMI).

It is one of the goals of the Foundational Subset for Executable UML Models specification to provide a true abstract base semantics for the foundation of UML.

6.4 Alignment with the OMG Four Layer Metamodeling Architecture

OMG modeling language specifications are developed within the framework of a four layer metamodeling architecture.

- M0-The domain under study (the “objects” of the model)
- M1-The user specification (the model)
- M2-The modeling language specification (the metamodel)
- M3-The reflexive metamodeling language specification (the meta-metamodel)

In terms of the OMG metamodeling layers, interpretation can generally be said to “cross meta-layers.” For example, the interpretation mapping for UML maps from model elements, considered to be “at layer M1,” to elements of the domain under study, considered to be “at layer M0.” Similarly, there are interpretation mappings from metamodel elements “at layer M2” to model elements “at layer M1” and from meta-metamodel elements “at layer M3” to metamodel elements “at layer M2.”

On the other hand, a theory is “within a single meta-layer.” For example, a theory of UML allows some models to be deduced from other models (e.g., instance models from class models), entirely at layer M1. Similarly, a theory of the UML abstract syntax allows the validity of a UML model to be determined entirely at level M2, after mapping the model to its metamodel representation.

Note that this view of the meta-layers does *not* consider elements in one layer to necessarily be “instances of” elements in the layer above it. For example, consider the particularly simple case of the domain at level M0 being Java programs. Typically, the relationship of the model of a class at level M1 to level M0 is considered to be something of the sort given in Figure 6.3.

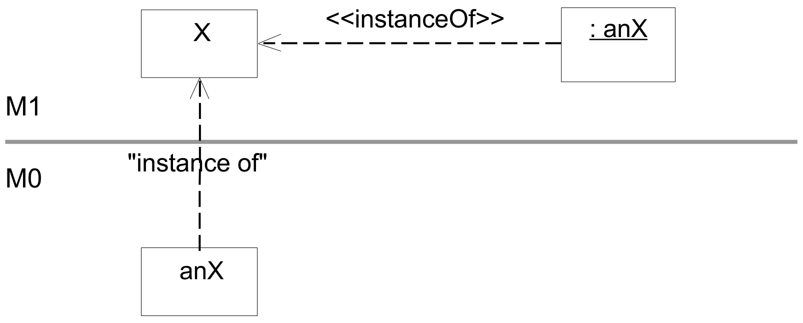


Figure 6.3 - Instance Relationship across Meta-Layers M1 and M0

The view taken here is that the concept of interpretation provides the general relationship between one meta-layer and the next. Thus, the above situation would be considered as in Figure 6.4. Despite the concrete example of a Java class used here at M0, the argument applies equally well to other more abstract domains, such as workers and the conceptual classes of their positions in a company.

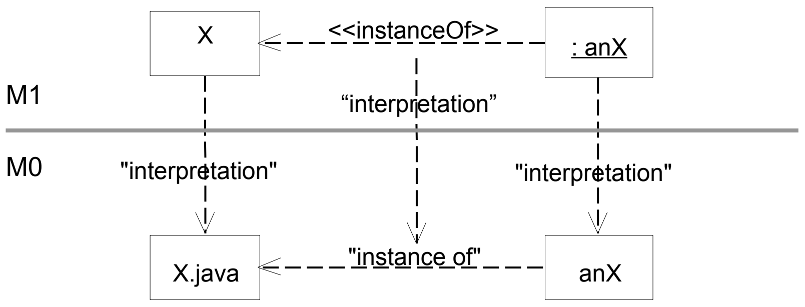


Figure 6.4 - Interpretation across Meta-Layers M1 and M0

If we now add level M2 to this diagram, the interpretation mapping is between instances of metaclasses at M2 and the model elements at M1. This is shown in Figure 6.5.

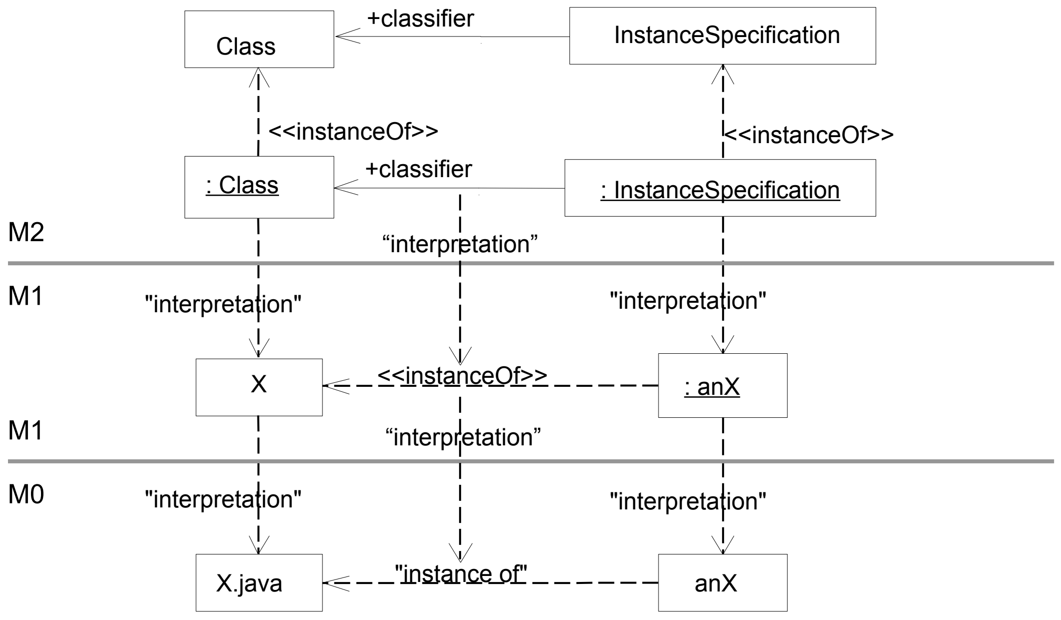


Figure 6.5 - Interpretation across Meta-Layers M2 and M1

The MOF takes the UML Infrastructure subset from layer M2 and places it in layer M3. The relationship between M2 and M3 is thus essentially the same as between M1 and M2. For example, the Class and InstanceSpecification metaclasses in layer M2 are represented as instances of the meta-metaclass Class in layer M3.

Now, it is common mental shorthand to identify a model element directly with its metamodel representation (e.g., the class X with its representation as an instance of the metaclass Class) and loosely refer to the model element as being directly "an instance of" the metaclass (e.g., class X "is an instance of" the metaclass Class). However, strictly speaking, the concept of "instance of" only has meaning within the theory of the metamodeling language. The fact that this concept is in the metamodeling language at all is merely consequence of the use in OMG of an object-oriented modeling language for metamodeling, which is not the only possible approach, and is not really fundamental to the relationship between the meta-layers.

6.5 Acknowledgments

The following companies submitted and/or supported parts of this specification.

6.5.1 Submitters

- CARE Technologies
- International Business Machines Corporation
- Kennedy Carter Ltd.
- Lockheed-Martin Corporation
- Mentor Graphics Corporation

- Model Driven Solutions

6.5.2 Supporters

- 88 Solutions Corporation
- CEA LIST
- NASA Jet Propulsion Laboratory
- U.S. National Institute of Standards and Technology

This page intentionally left blank

7 Abstract Syntax

7.1 Overview

This clause defines the subset of UML for which foundational semantics are specified in Clause 8. This subset is called Foundational UML or fUML. It is a computationally complete language for executable models.

A fundamental purpose of fUML is to serve as an intermediary between "surface subsets" of UML used for modeling and computational platform languages used as the target for model execution. As shown in Figure 7.1, this generally requires the ability to translate from the surface subset to fUML and from fUML to the target platform language.

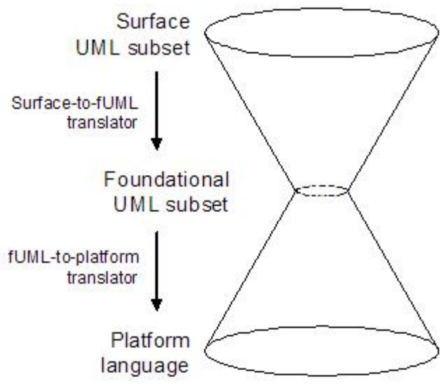


Figure 7.1 - Translation to and from the foundational UML subset

In this context, the contents of the fUML subset has been largely determined by three criteria.

- Compactness – The subset should be small to facilitate definition of a clear semantics and implementation of execution tools.
- Ease of translation – The subset should enable straightforward translation from common surface subsets of UML to fUML and from fUML to common computational platform languages.
- Action functionality – This specification only specifies how to execute the UML actions as they are currently defined with primitive functionality. Therefore, the fUML subset should not include UML functionality requiring coordinated sets of UML actions to reproduce.

There is, of course, some tension between these criteria.

Suppose that there is a surface feature of UML (say, polymorphic operation dispatching) that also happens to have a corresponding analog in a certain platform language (say, an object-oriented programming language such as Java), but which is excluded from fUML (though, in this case, it actually isn't). It is clearly desirable that the surface UML feature be translated, ultimately, into the corresponding feature of the platform language. However, if the feature is excluded from fUML, it is necessary for the surface-to-fUML translator to generate a coordinated set of fUML elements that has the same effect as that feature. But then the fUML-to-platform translator would need to recognize the pattern generated by the surface-to-fUML generator, in order to map this back into the desired feature of the target language. Compactness can therefore conflict with ease of translation.

Unfortunately, in practice, such overlaps between desired features in the surface subset of UML used for modeling and the available features of the target platform language can be significant, especially within a single domain of application. Further, the specific pattern of elements that might be generated by a surface-to-fUML translator for any given surface feature is not standardized-and such a standard is not in the scope of this specification. Therefore, a general fUML-to-platform translator cannot be optimized to specially handle a standard set of expected patterns.

On the other hand, if a feature of UML is included in fUML to reduce the translation problems described above, it increases the complexity of the semantics of fUML and the implementation of execution tools conforming to those semantics. This might not be so bad for any individual feature, but an accumulation of many such features will eventually defeat the purpose of having a compact subset.

The subset specified in this clause resolves the choice between compactness and ease of translation based on judgments about which functionalities in common between UML and computational platforms are more widely used than others. These judgments have the hazard of making broad generalizations about highly segmented modeling and platform markets, but once made, they help determine the contents of the foundational subset as follows:

- Widely used functionality in common between UML and platforms should have the simplest translation into and out of the fUML subset, namely, one-to-one translations. This functionality is included in the foundational subset. For example, classes with properties and operations are widely used elements of object oriented models and control and object flows are widely used in activity modeling.
- Moderately used functionality in common between UML and platforms should have a straightforward translation into and out of the foundational subset. This translation is not one-to-one, so this functionality is not included in the fUML subset, but the elements needed to enable straightforward mappings are included. For example, composite structure and simple state machines are considered moderately used.
- Less used functionality in common between UML and platforms may have a complicated translation into the fUML subset and is not included in the foundational subset. Little consideration is given to including functionality to simplify the translation. For example, association qualifiers and interruptible activity regions are considered less used.

Further, certain modeling features of UML are not directly supported by UML action functionality. For example, the UML semantics of default attribute values is that the default values are assigned to attributes when the object is created. However, the UML semantics for create object actions require that objects be created without attribute values being set. Therefore, making the semantics of UML default values explicit requires coordinated actions for creating objects and assigning structural feature values, with activity control and object flows between them. Consequently, default attribute values are not included in the foundational subset. In cases such as this, it is expected that the transformational approach above will be used to generate the set of actions corresponding to desired surface UML semantics. (Note, for example, that this is particularly important for embedded systems, where the execution of default actions for initialization purposes must carefully coordinate with other initialization activities.)

Finally, the fUML subset also contains some UML elements that have no execution semantics. Examples of this are comments and packages from Kernel and modeling declarations such as isDeterminate and isAssured on conditional nodes. These reduce compactness of the subset but not in a way that affects the specification of semantics, the implementation of execution tools or translator construction.

7.2 Syntax Packages

Subclauses 7.3 and following define the abstract syntax of the fUML subset as a subset of the abstract syntax of UML 2. The package structure parallels the package structure of the UML 2 abstract syntax model. Packages in the UML 2 model that have no corresponding package here are excluded in their entirety. For packages that are included, some further elements from the UML package may be excluded in the corresponding fUML package. In this case, the model presented for the fUML version of the abstract syntax package shows those elements that are specifically in fUML and specify additional constraints on the class that that apply in addition to the constraints already specified in UML 2 for the same class.

The fUML subset definition is formally captured in the package fUML_Syntax::Syntax. This package includes the subpackages shown in Figure 7.2, each of which imports into its namespace exactly those metaclasses from the corresponding UML abstract syntax package. All the elements in the subpackages are then re-imported into the top-level Syntax package, which allows them to be uniformly referenced by qualified name directly from the top-level package (similarly to the namespace structure used in the UML abstract syntax metamodel).

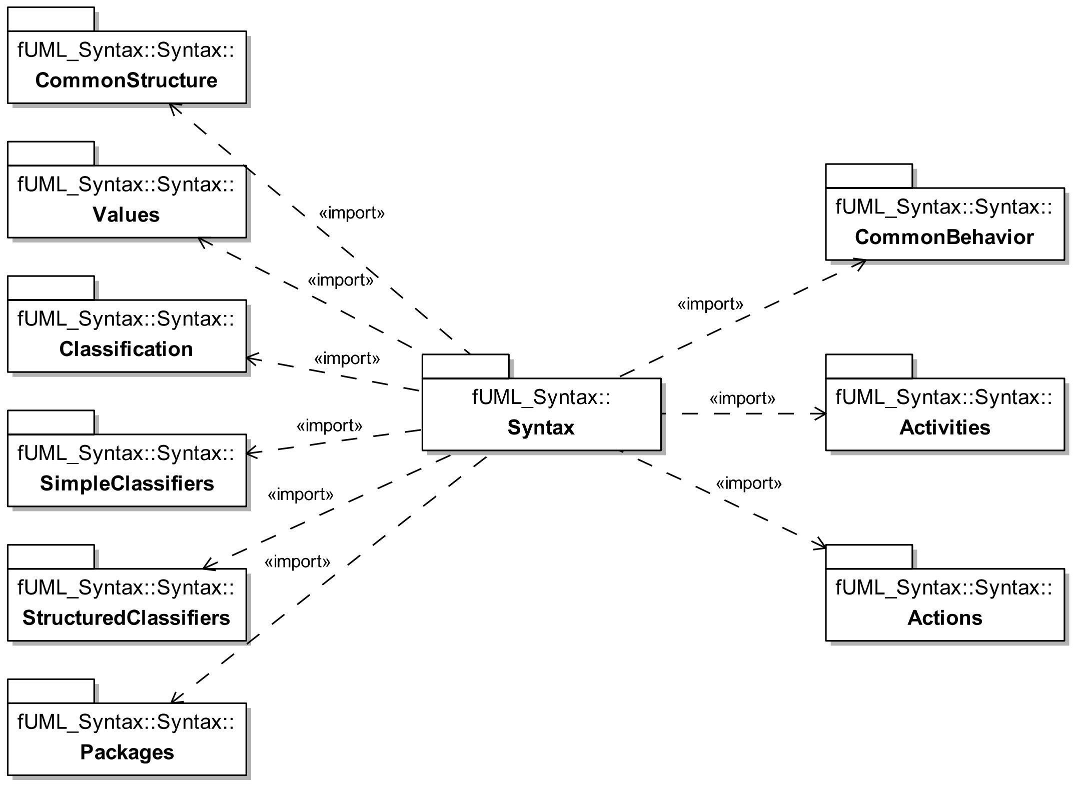


Figure 7.2 - fUML Syntax Package

A UML model that syntactically conforms to this subset shall have an abstract syntax representation that consists solely of instances of metaclasses that are (imported) members of the fUML_Syntax::Syntax package. For simplicity, meta-associations from the UML abstract syntax metamodel are *not* explicitly imported into the fUML_Syntax::Syntax package, but it is, nevertheless, permissible for the model elements of a conforming model, within the fUML subset, to be involved in any meta-associations consistent with both the UML metamodel and any further constraints as defined in this specification.

Note. This approach for defining a subset of the UML abstract syntax is similar to the approach used for defining the metamodel subset covered by a UML profile, in which specially identified package imports (metamodelReferences) and element imports (metaclassReferences) are used to import the metaclasses from the subset into the namespace of the Profile (see the UML 2 Specification, 12.3).

In addition to being representable within the fUML abstract syntax subset, as described above, a UML model that syntactically conforms to fUML shall also satisfy all relevant constraints defined in the UML abstract syntax metamodel *and* the additional syntactic constraints specified here for fUML. The fUML semantics specified in Clause 8 are only defined for well-formed fUML models that meet all the necessary constraints.

The constraints specified for fUML are all those that are imported members of the fUML_Syntax::Constraints package (see Figure 7.3). Each of these constraints has as its single constrained element the UML abstract syntax metaclass to which the constraint applies. The constraints are organized into subpackages, similarly to the those in the Syntax subset model, and then re-imported into the top-level Constraints package. (However, there are no additional constraints for the Values and Packages syntactic packages, so there are no corresponding Constraint subpackages for these.)

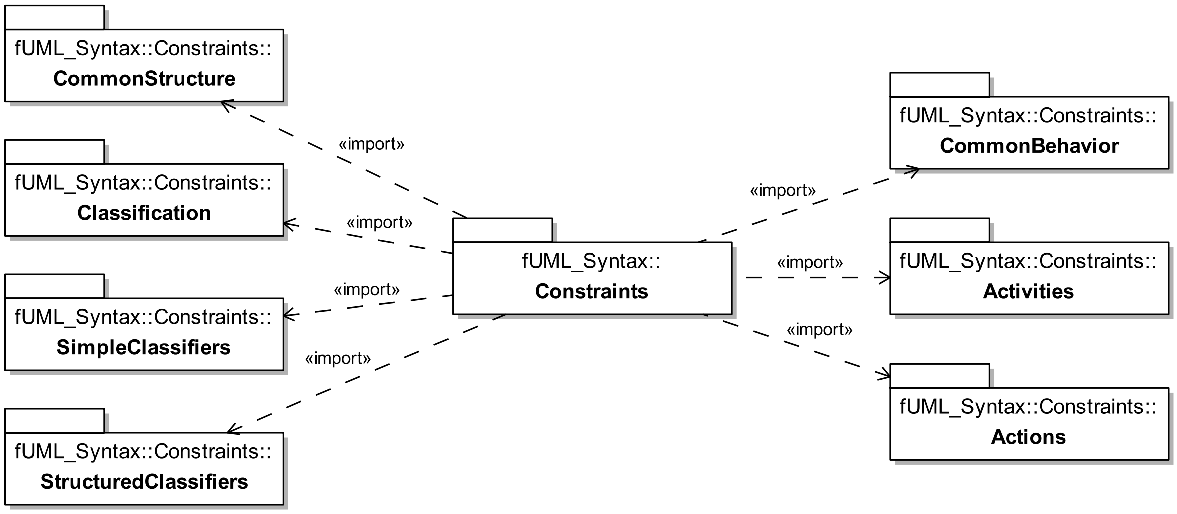

fUML_Syntax::Constraints::
CommonStructure
fUML_Syntax::Constraints::
Classification
fUML_Syntax::Constraints::
SimpleClassifiers
fUML_Syntax::Constraints::
StructuredClassifiers
fUML_Syntax::
Constraints
«import»
«import»
«import»
«import»
«import»
«import»
«import»
fUML_Syntax::Constraints::
CommonBehavior
fUML_Syntax::Constraints::
Activities
fUML_Syntax::Constraints::
Actions

Figure 7.3 - fUML Constraints Package

7.3 Common Structure

7.3.1 Overview

The fUML CommonStructure package imports classes from the UML CommonStructure package. The classes shown in Figures 7.4 to 7.6 are those included in the fUML CommonStructure package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Root (see Figure 7.4): No exclusions.

From Templates (no corresponding fUML diagram):

- All classes related to templates are excluded from fUML, because templates are outside the scope of the fUML specification.

From Namespaces (see Figure 7.5):

- Namespace::nameExpression – This is excluded because name expressions are used to provide computed names within templates, and templates are excluded from the fUML subset.
- Namespace::ownedRule – Namespaces cannot own constraints in fUML, because constraints are excluded from the fUML subset.

From Types and Multiplicities (see Figure 7.6): No exclusions

From Constraints (no corresponding fUML diagram):

- Constraint – Constraints are excluded from fUML, because they are considered to be design-time annotations that should already be satisfied by a well-formed model. Otherwise, the general semantics of the run time checking of constraints is not currently well specified in UML 2, particularly when constraints should be evaluated and what should happen if they should fail. Further elaboration of the semantics of constraint checking in UML was judged to be outside the scope of the fUML specification.

From Dependencies (no corresponding fUML diagram):

- All classes related to dependencies are excluded from fUML, because dependencies either declare a design intent or express a model-level relationship without significant execution semantics.

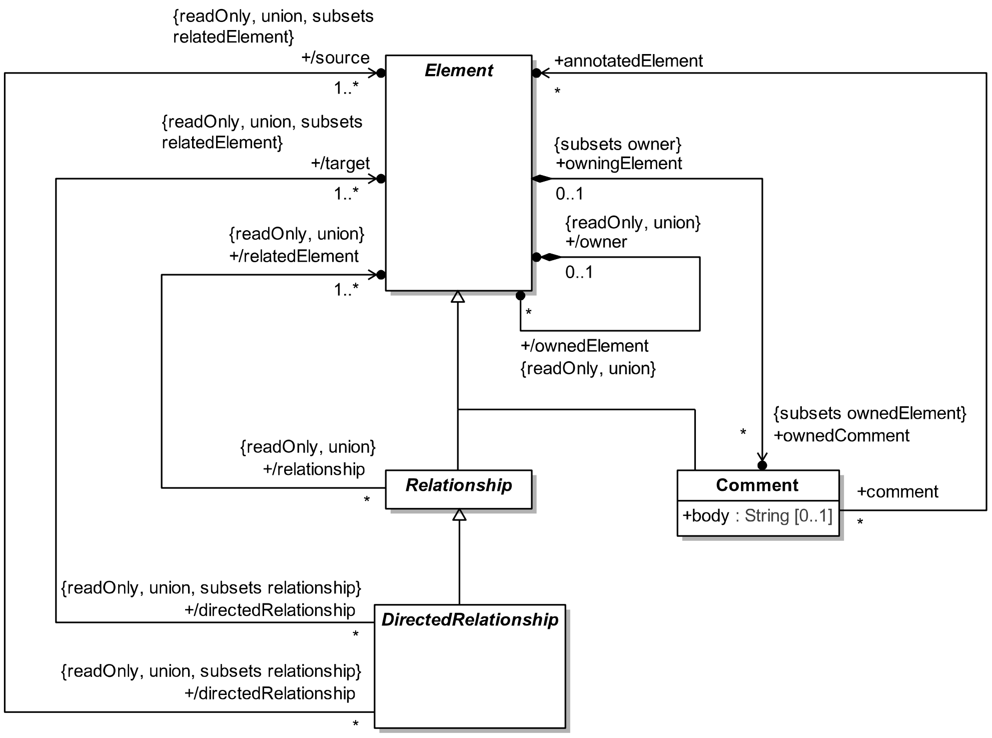


Figure 7.4 - Root

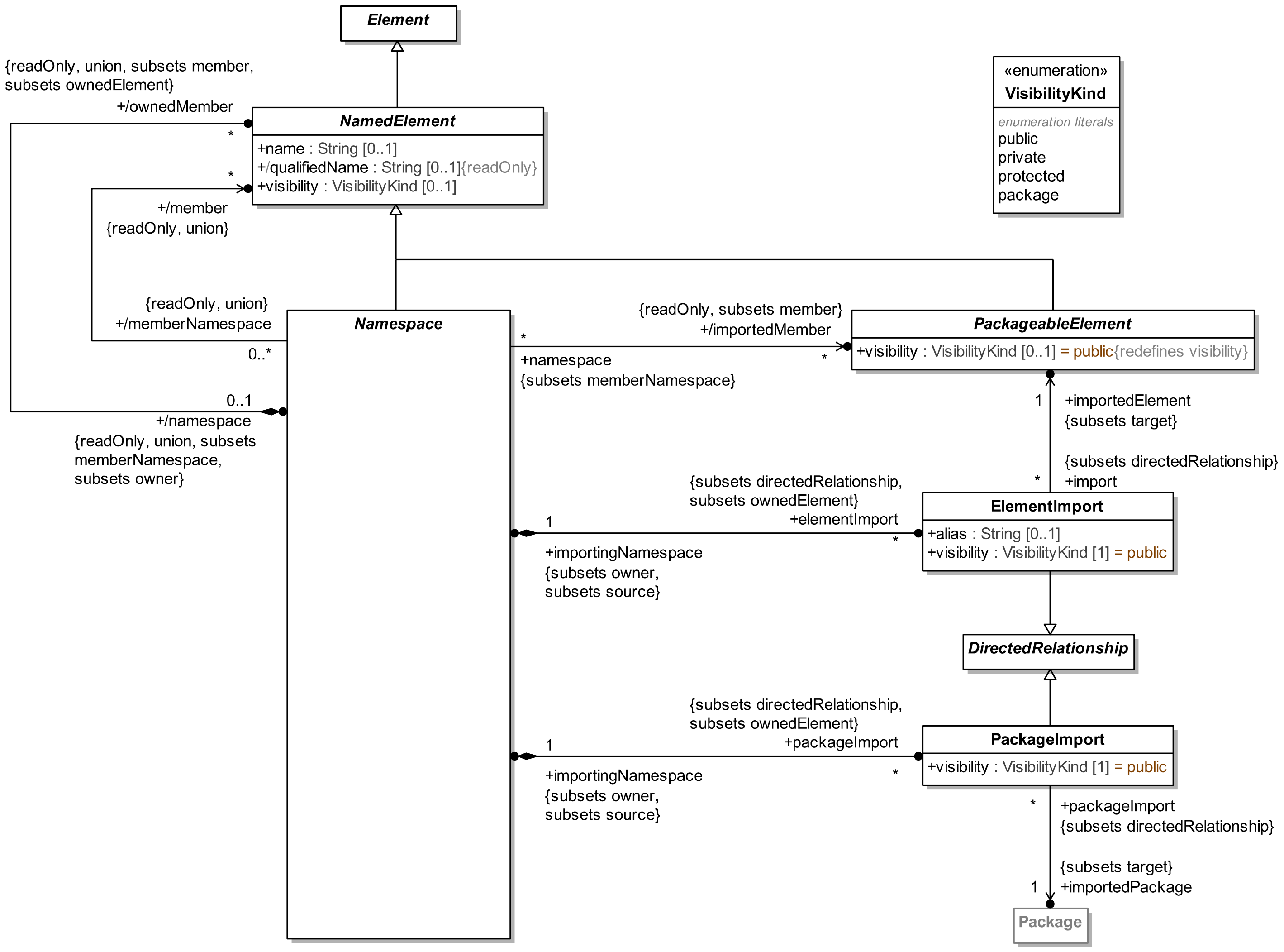

Element
NamedElement
+name : String [0..1]
+/qualifiedName : String [0..1]{readOnly}
+visibility : VisibilityKind [0..1]
{readOnly, union, subsets member, subsets ownedElement}
+/ownedMember
*
*
+/member
{readOnly, union}
«enumeration»
VisibilityKind
enumeration literals
public
private
protected
package
{readOnly, union}
+/memberNamespace
0..*
Namespace
0..1
+/namespace
{readOnly, union, subsets memberNamespace, subsets owner}
{readOnly, subsets member}
+/importedMember
*
*
+namespace
{subsets memberNamespace}
PackageableElement
+visibility : VisibilityKind [0..1] = public{redefines visibility}
1
+importedElement
{subsets target}
{subsets directedRelationship}
+import
*
{subsets directedRelationship, subsets ownedElement}
+elementImport
1
*
ElementImport
+alias : String [0..1]
+visibility : VisibilityKind [1] = public
+importingNamespace
{subsets owner, subsets source}
DirectedRelationship
{subsets directedRelationship, subsets ownedElement}
+packageImport
1
*
PackageImport
+visibility : VisibilityKind [1] = public
+importingNamespace
{subsets owner, subsets source}
*
+packageImport
{subsets directedRelationship}
{subsets target}
+importedPackage
1
Package

Figure 7.5 - Namespaces

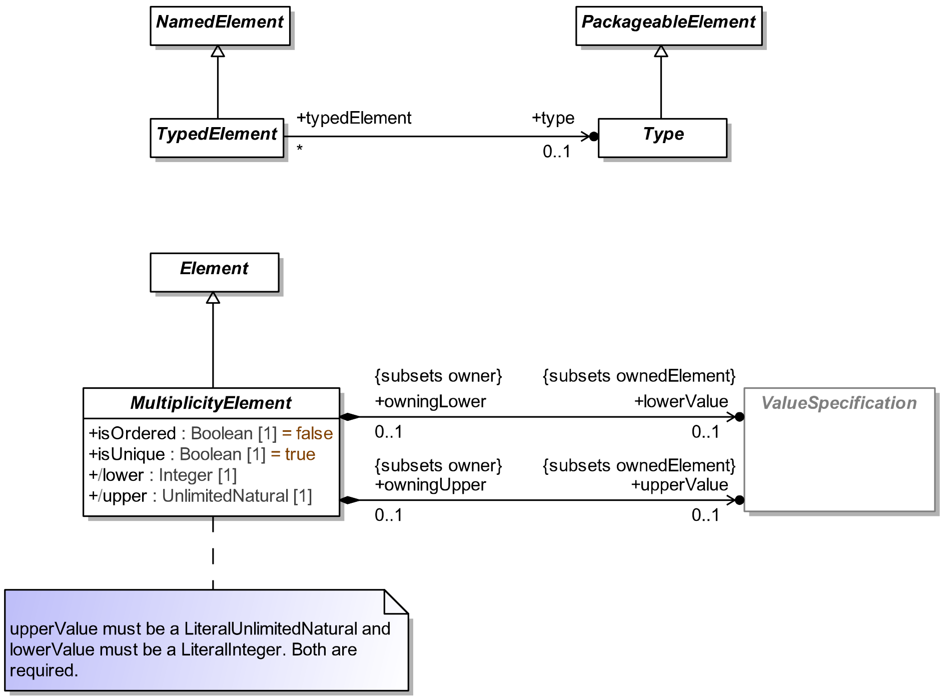


Figure 7.6 - Types and Multiplicity

7.3.2 Constraints

7.3.2.1 MultiplicityElement

[1] fuml_multiplicity_element_required_lower_and_upper
upperValue must be a LiteralUnlimitedNatural and lowerValue must be a LiteralInteger. Both are required.

```
self.upperValue->notEmpty() and
self.upperValue->asSequence()->first().oclIsKindOf(LiteralUnlimitedNatural) and
self.lowerValue->notEmpty() and
self.lowerValue->asSequence()->first().oclIsKindOf(LiteralInteger)
```

7.4 Values

7.4.1 Overview

The fUML Values package imports classes from the UML Values package. The classes shown in Figure 7.7 are those included in the fUML Values package. The diagram corresponds to a similar diagrams in the UML 2 Specification. The following classes have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Literals (see Figure 7.7): No exclusions.

From Expressions (no corresponding fUML diagram):

- Expression – Expressions are excluded from fUML because, in UML, this construct simply captures the parse tree of an expression whose symbols are otherwise only informally represented as strings and thus cannot be properly executed.
- OpaqueExpression – Opaque expressions are excluded from fUML because their body is not further defined within UML and, thus, not executable. Opaque expressions also allow for an optional association with a UML behavior. However, this was considered to be redundant with the ability to directly call behaviors within the context of UML activities, the primary form of behavior modeling supported in fUML.
- StringExpression – String expressions are excluded from fUML because they are only used within the context of templates, which are outside the scope of fUML.

From Time and Duration (no corresponding fUML diagram):

- Time values are excluded entirely from fUML because time events and constraints are not within the scope of fUML.

From Intervals (no corresponding fUML diagram):

- Intervals and interval constraint are excluded entirely from fUML because they are are outside the scope of fUML.

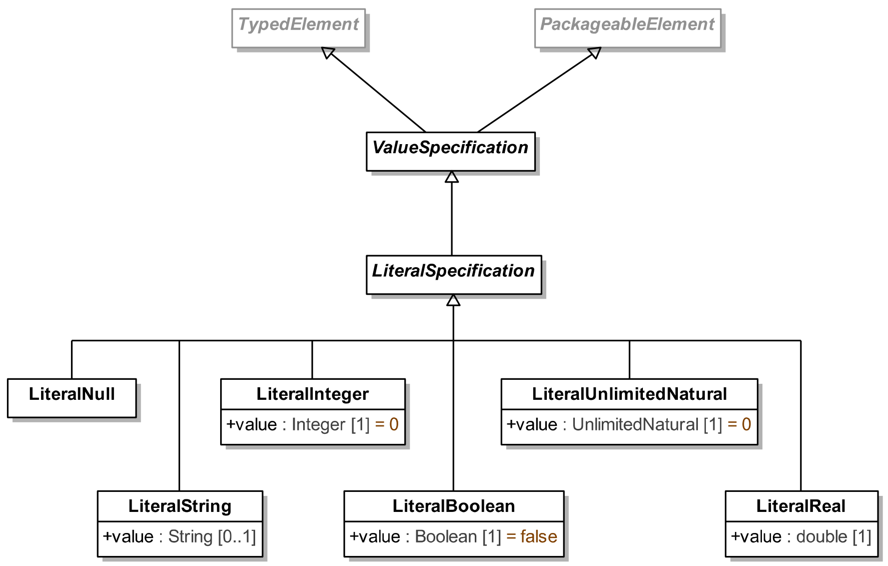


Figure 7.7 - Literals

7.4.2 Constraints

None.

7.5 Classification

7.5.1 Overview

The fUML Classification package imports classes from the UML Classification package. The classes shown in Figures 7.8 to 7.12 are those included in the fUML Classification package. The diagrams correspond to similar diagrams in the UML 2

Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Classifiers (see Figure 7.8):

- Classifier::redefinedClassifier – Classifier redefinition is excluded from fUML because it was judged to add significant complexity to resolve during execution, without a fundamental need in the majority of cases.
- Classifier::powerTypeExtent – This is excluded because generalization sets not included in fUML.
- Classifier::collaborationUse and Classifier::representation – These are excluded because collaborations are outside the of scope of fUML.
- Classifier::useCase – This is excluded because use cases are outside the scope of fUML.
- Classifier::substitution – This is excluded because substitution dependencies are not included in fUML.

From Classifier Templates (no corresponding fUML diagram):

- All classes related to classifier templates are excluded from fUML because templates are outside the scope of fUML.

From Features (see Figure 7.9):

- BehavioralFeature::ownedParameterSet – This is excluded because parameter sets are not included in fUML.
- Parameter::defaultValue – Implicitly computing a default value for a behavioral feature (or behavior) would require coordination of multiple UML actions, since call actions always require explicit inputs or outputs to be provided.
- Parameter::parameterSet – This is excluded because parameter sets are not included in fUML.

From Properties (see Figure 7.10):

- Property::defaultValue – Setting defaults requires coordination of multiple UML actions, since the create object action is specified to create objects without default values. Setting defaults in fUML must be modeled explicitly by using the appropriate structural feature actions after object creation.
- Property::qualifier – Association qualifiers are excluded from fUML because their effect can be effectively achieved in models using unqualified associations and so are not considered fundamental. Further, they were judged not widely used enough to otherwise require inclusion in fUML for ease of implementation of execution tools and translators.
- Property::subsettedProperty – Subsetting is excluded from fUML because subsetting is generally used in static models and there is no consensus on the execution semantics for this mechanism. (See 8.1 for further discussion of conventions related to the handling of subsetting in fUML execution semantics.)
- Property::redefinedProperty – Property redefinition is excluded from fUML because it was judged to add significant complexity to the resolution of structural features at runtime, without a fundamental need. (Note, on the other hand, that operation redefinition is included in fUML, as shown in Figure 7.11, because it is necessary for the default fUML semantics for polymorphic operation dispatching, as discussed in 8.7.1. Also see 8.1 for further discussion of conventions related to the handling of redefinition in fUML execution semantics.)
- Property::interface – This is excluded because interfaces are outside the scope of fUML.

From Operations (see Figure 7.11):

- Operation::raisedException – This is excluded because exceptions are not included in fUML.
- Operation::templateParameter – This is excluded because templates are outside the scope of fUML.

- Operation::precondition, Operation::postcondition and Operation::bodyCondition – These are excluded because constraints are not included in fUML.
- Operation::interface – This is excluded because interfaces are outside the scope of fUML.
- Operation::datatype – This is excluded because data types cannot have operations in fUML.

From Generalization Sets (no corresponding fUML diagram):

- Power types and generalization sets add significant complexity to the semantics of generalization, particularly as it relates to typing and polymorphic operation dispatching. Further, the effect of a generalization set can be equivalently modeled using regular classes and generalizations, albeit at the expense of some modeling convenience. Power types and generalization sets are therefore not considered fundamental for the fUML subset.

From Instances (see Figure 7.12):

- InstanceSpecification::specification – Instance specifications in fUML are only used as part of the value specification of a structured instance value, which is specified using slots, or as an enumeration literal (see Figure 7.13). Therefore, it is unnecessary to provide a separate specification for its value.

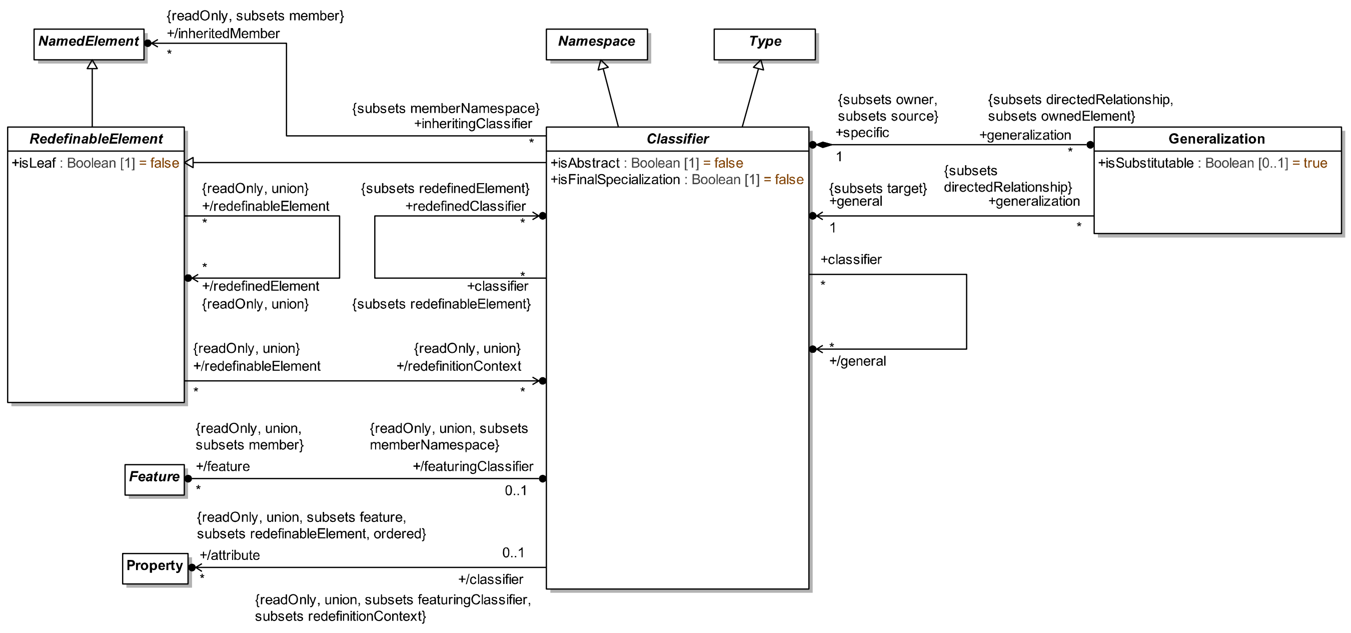
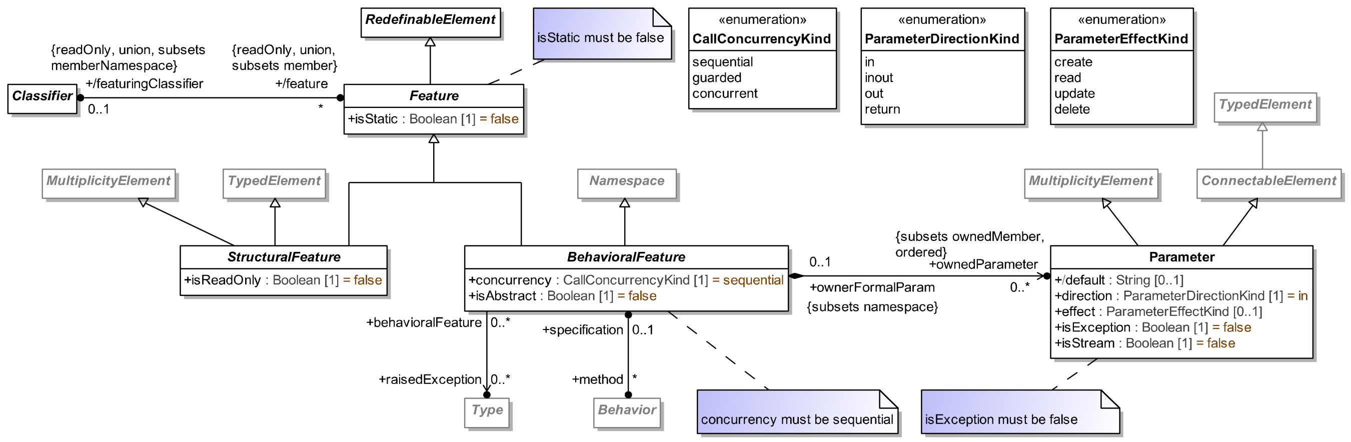
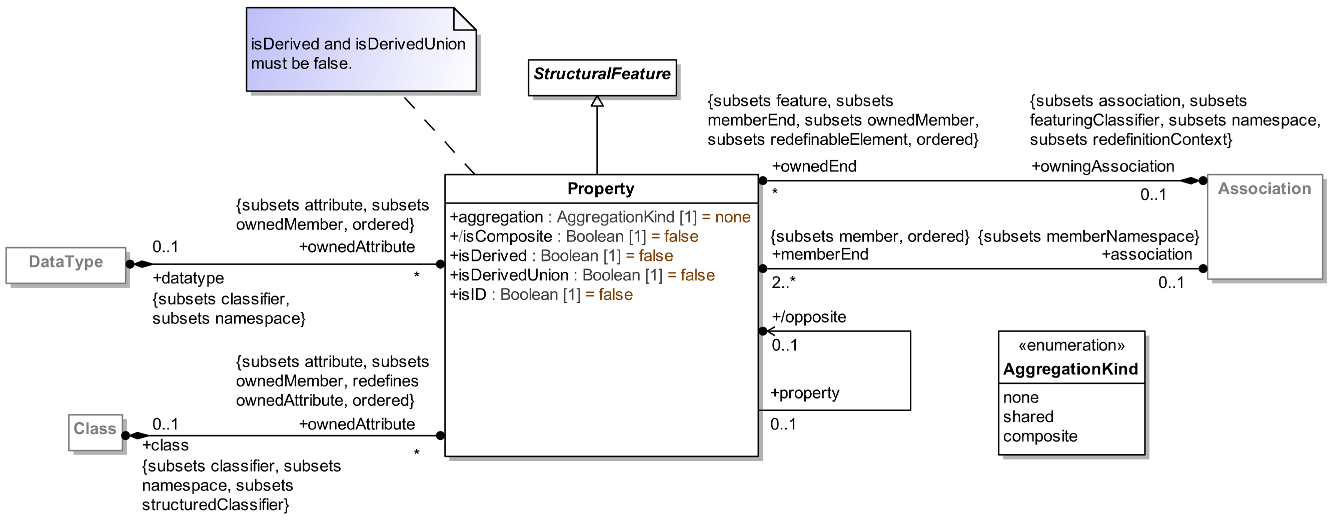

NamedElement
{readOnly, subsets member}
+/inheritedMember
Namespace
Type
RedefinableElement
+isLeaf : Boolean [1] = false
{subsets memberNamespace}
+inheritingClassifier
Classifier
+isAbstract : Boolean [1] = false
+isFinalSpecialization : Boolean [1] = false
{subsets owner, subsets source}
+specific
{subsets directedRelationship, subsets ownedElement}
+generalization
Generalization
+isSubstitutable : Boolean [0..1] = true
{subsets target}
+general
{subsets directedRelationship}
+generalization
{readOnly, union}
+/redefinableElement
{subsets redefinedElement}
+redefinedClassifier
+/redefinedElement
{readOnly, union}
+classifier
{subsets redefinableElement}
+classifier
+/general
{readOnly, union}
+/redefinableElement
{readOnly, union}
+/redefinitionContext
Feature
{readOnly, union, subsets member}
+/feature
{readOnly, union, subsets memberNamespace}
+/featuringClassifier
0..1
Property
{readOnly, union, subsets feature, subsets redefinableElement, ordered}
+/attribute
0..1
+/classifier
{readOnly, union, subsets featuringClassifier, subsets redefinitionContext}

Figure 7.8 - Classifiers



RedefinableElement
isStatic must be false
«enumeration» CallConcurrencyKind
sequential
guarded
concurrent
«enumeration» ParameterDirectionKind
in
inout
out
return
«enumeration» ParameterEffectKind
create
read
update
delete
{readOnly, union, subsets memberNamespace}
+/featuringClassifier
{readOnly, union, subsets member}
+/feature
Classifier
0..1
*
Feature
+isStatic : Boolean [1] = false
TypedElement
MultiplicityElement
TypedElement
Namespace
MultiplicityElement
ConnectableElement
StructuralFeature
+isReadOnly : Boolean [1] = false
BehavioralFeature
+concurrency : CallConcurrencyKind [1] = sequential
+isAbstract : Boolean [1] = false
{subsets ownedMember, ordered}
+ownedParameter
0..1
+ownerFormalParam
0..*
{subsets namespace}
Parameter
+/default : String [0..1]
+direction : ParameterDirectionKind [1] = in
+effect : ParameterEffectKind [0..1]
+isException : Boolean [1] = false
+isStream : Boolean [1] = false
+behavioralFeature
0..*
+specification
0..1
+raisedException
0..*
+method
*
Type
Behavior
concurrency must be sequential
isException must be false

Figure 7.9 - Features



isDerived and isDerivedUnion must be false.
StructuralFeature
{subsets feature, subsets memberEnd, subsets ownedMember, subsets redefinableElement, ordered}
+ownedEnd
*
{subsets association, subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+owningAssociation
0..1
Association
Property
+aggregation : AggregationKind [1] = none
+/isComposite : Boolean [1] = false
+isDerived : Boolean [1] = false
+isDerivedUnion : Boolean [1] = false
+isID : Boolean [1] = false
{subsets attribute, subsets ownedMember, ordered}
0..1
+ownedAttribute
DataType
+datatype
*
{subsets classifier, subsets namespace}
{subsets member, ordered}
+memberEnd
2..*
{subsets memberNamespace}
+association
0..1
+/opposite
0..1
+property
0..1
«enumeration» AggregationKind
none
shared
composite
{subsets attribute, subsets ownedMember, redefines ownedAttribute, ordered}
0..1
+ownedAttribute
Class
+class
*
{subsets classifier, subsets namespace, subsets structuredClassifier}

Figure 7.10 - Properties

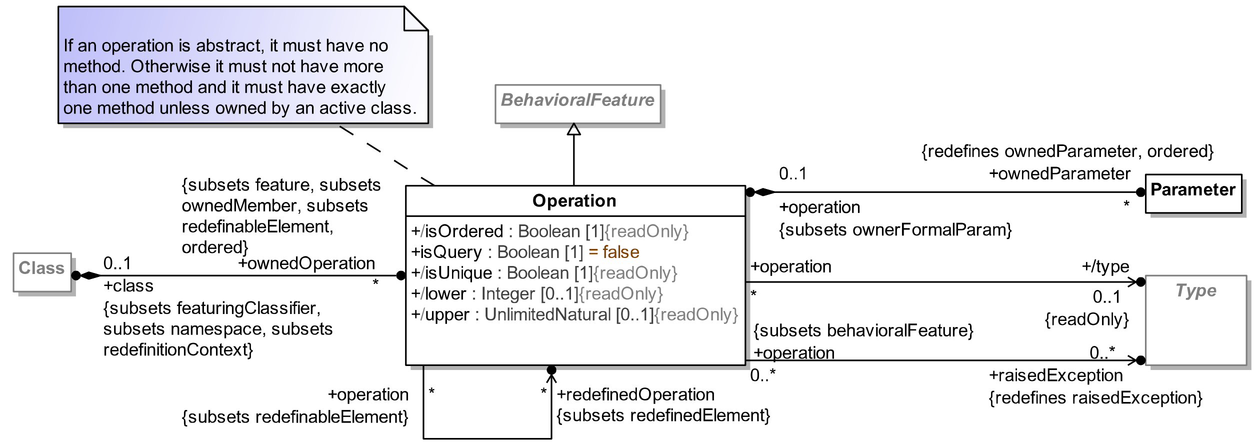

If an operation is abstract, it must have no method. Otherwise it must not have more than one method and it must have exactly one method unless owned by an active class.
BehavioralFeature
Operation
+/isOrdered : Boolean [1]{readOnly}
+isQuery : Boolean [1] = false
+/isUnique : Boolean [1]{readOnly}
+/lower : Integer [0..1]{readOnly}
+/upper : UnlimitedNatural [0..1]{readOnly}
Class
0..1
+class
{subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
{subsets feature, subsets ownedMember, subsets redefinableElement, ordered}
+ownedOperation
*
{redefines ownedParameter, ordered}
+ownedParameter
0..1
+operation
{subsets ownerFormalParam}
Parameter
+operation
*
+/type
0..1
{readOnly}
Type
{subsets behavioralFeature}
+operation
0..*
0..*
+raisedException
{redefines raisedException}
+operation
{subsets redefinableElement}
+redefinedOperation
{subsets redefinedElement}

Figure 7.11 - Operations

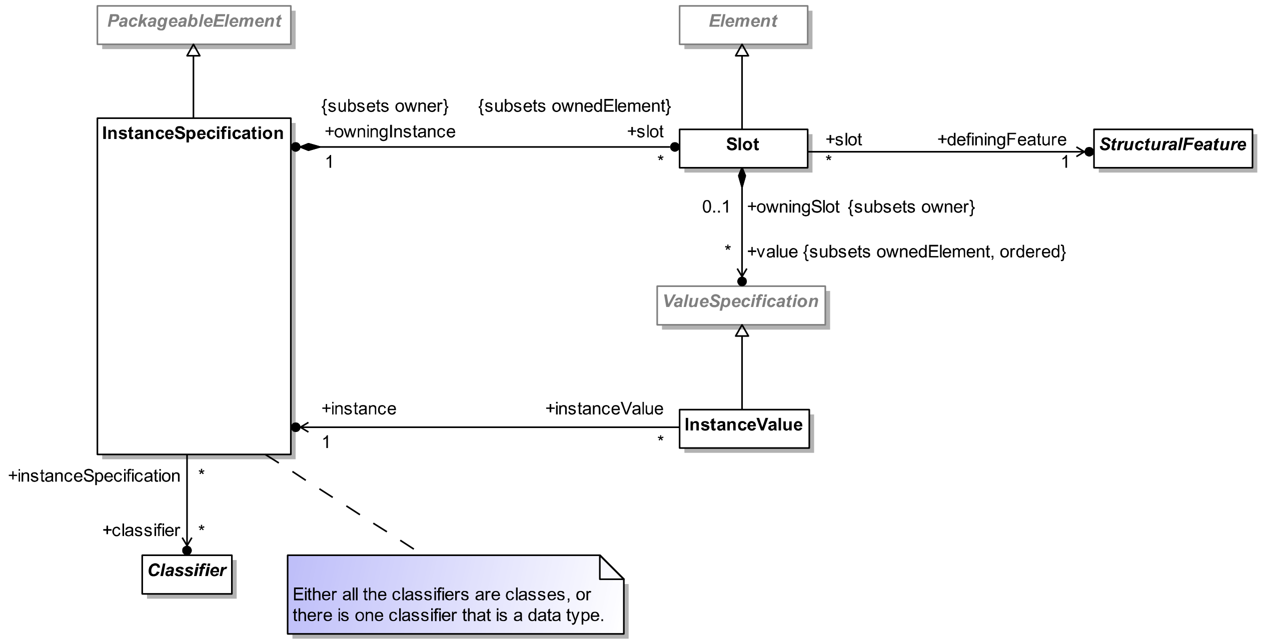


Figure 7.12 - Instances

7.5.2 Constraints

7.5.2.1 BehavioralFeature

[1] fuml_behavioral_feature_sequentiality
concurrency must be sequential

self.concurrency = CallConcurrencyKind::sequential

7.5.2.2 Feature

[1] fuml_feature_non_static
isStatic must be false

```
not self.isStatic
```

7.5.2.3 InstanceSpecification

[1] fuml_instance_specification_possible_classifiers
Either all the classifiers are classes, or there is one classifier that is a data type

```
self.classifier->forAll(oclIsKindOf(Class)) or
self.classifier->size() = 1 and self.classifier->forAll(oclIsKindOf(DataType))
```

7.5.2.4 Operation

[1] fuml_operation_has_at_most_one_method
If an operation is abstract, it must have no method. Otherwise it must not have more than one method and it must have exactly one method unless owned by an active class.

```
If self.isAbstract then self.method->isEmpty()
else
   self.method->size() <= 1 and
   ((self.class = null or not self.class.isActive) implies
      self.method->size() = 1)
endif
```

7.5.2.5 Parameter

[1] fuml_parameter_not_exception
isException must be false

```
not self.isException
```

7.5.2.6 Property

[1] no_derivation
isDerived and isDerivedUnion must be false

```
not self.isDerived and not self.isDerivedUnion
```

7.6 Simple Classifiers

7.6.1 Overview

The fUML SimpleClassifiers package imports classes from the UML SimpleClassifiers package. The classes shown in Figures 7.13 to 7.15 are those included in the fUML SimpleClassifiers package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Data Types (see Figure 7.13):

- DataType::ownedOperation – Data types cannot have operations in fUML because they are not behaviored classifiers and so cannot own behaviors. This means that there is no way to provide executable methods for the operations of a data type.

From Signals (see Figure 7.14): No exclusions.

From Interfaces (see Figure 7.15):

- Interfaces – Within the fUML subset, the effect of interfaces can be achieved using abstract classes with entirely abstract operations. (Note that fUML does not include UML 2 structured classes and ports, which depend specifically on the use of interfaces.)
- InterfaceRealization – This is excluded because interfaces are not included in fUML.
- BehavioredClassifier::interfaceRealization – This is excluded because interface realizations are not included in fUML.

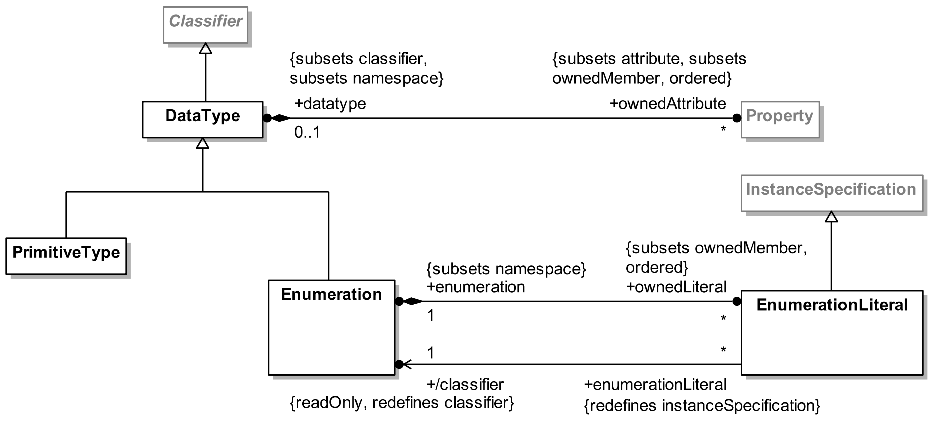


Figure 7.13 - Data Types

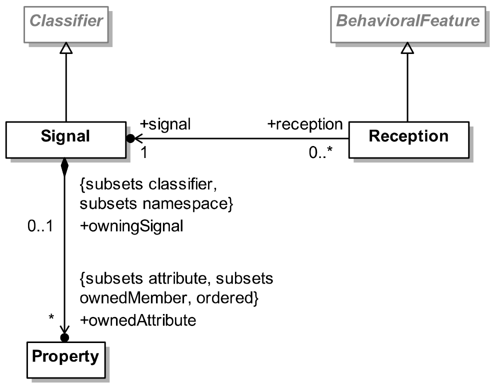


Figure 7.14 - Signals

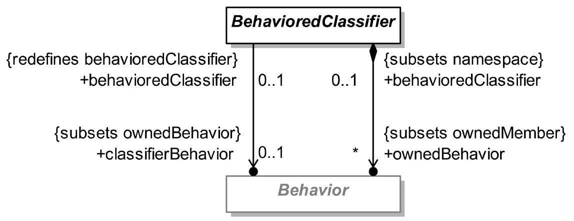


Figure 7.15 - Interfaces

7.6.2 Constraints

7.6.2.1 Reception

[1] fuml_reception_no_method
A reception must not have an associated method.

```
self.method->isEmpty()
```

[2] fuml_reception_not_abstract
A reception must not be abstract.

```
not self.isAbstract
```

7.7 Structured Classifiers

7.7.1 Overview

The fUML StructuredClassifiers package imports classes from the UML StructuredClassifiers package. The classes shown in Figures 7.16 to 7.17 are those included in the fUML StructuredClassifiers package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Structured Classifiers (no corresponding fUML diagram):

- Connectors are entirely excluded as being outside the scope of fUML.

From Encapsulated Classifiers (no corresponding fUML diagram):

- Ports are entirely excluded as being outside the scope of fUML.

From Classes (see Figure 7.16):

- Class::extension – This is excluded because stereotypes are outside the scope of fUML.

From Associations (see Figure 7.17):

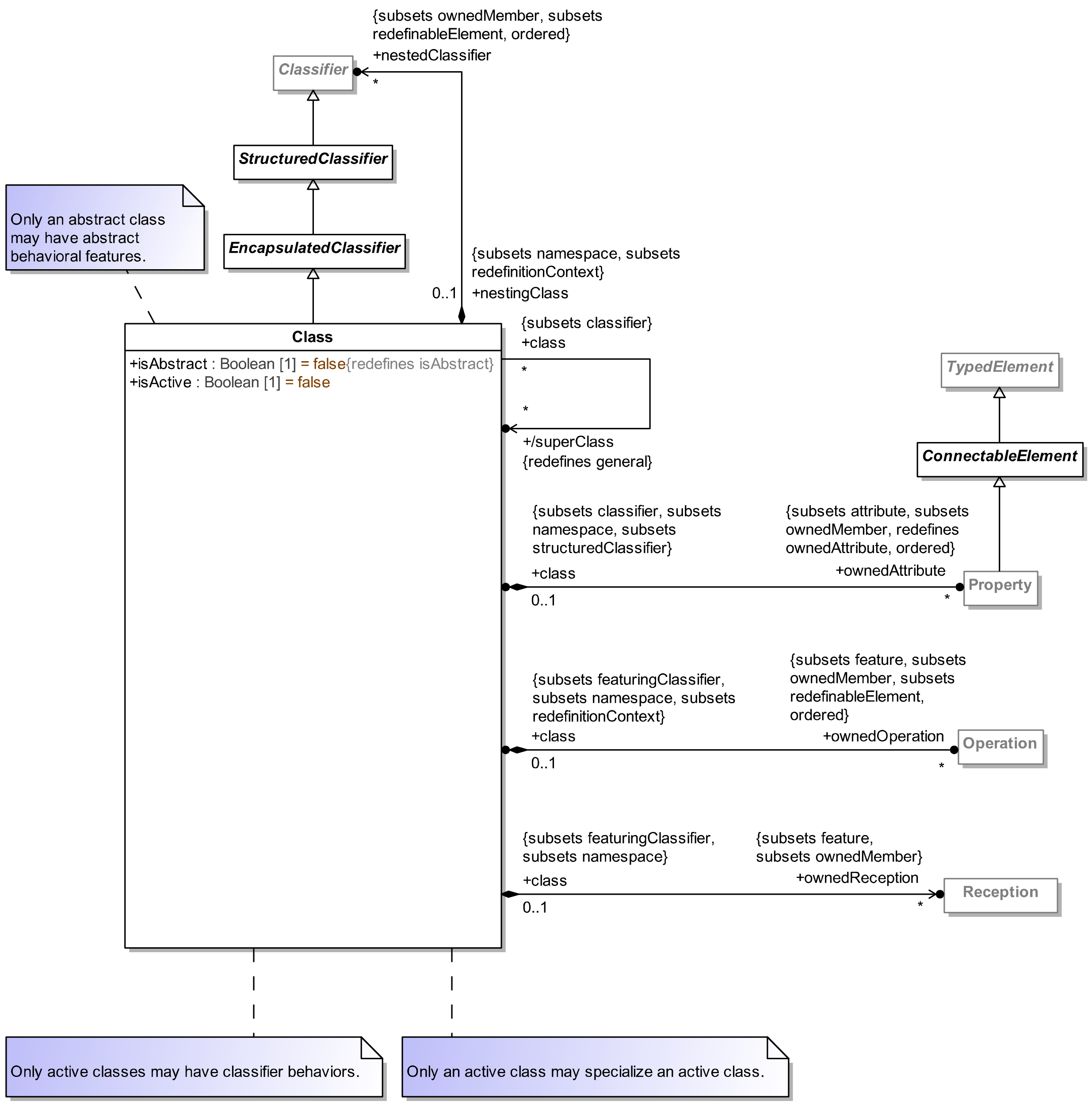
- AssociationClass – Association classes, as a modeling construct, add significant semantic complexity and their effect can be equivalently modeled using regular classes and associations, albeit at the expense of some modeling convenience. They are therefore not considered fundamental for the fUML subset.

From Components (no corresponding fUML diagram):

- Components are entirely excluded as being outside the scope of fUML.

From Collaborations (no corresponding fUML diagram):

- Collaborations are entirely excluded as being outside the scope of fUML.



{subsets ownedMember, subsets redefinableElement, ordered}
+nestedClassifier
*
Classifier
StructuredClassifier
EncapsulatedClassifier
Only an abstract class may have abstract behavioral features.
{subsets namespace, subsets redefinitionContext}
0..1
+nestingClass
Class
+isAbstract : Boolean [1] = false{redefines isAbstract}
+isActive : Boolean [1] = false
{subsets classifier}
+class
*
*
+/superClass
{redefines general}
TypedElement
ConnectableElement
{subsets classifier, subsets namespace, subsets structuredClassifier}
+class
0..1
{subsets attribute, subsets ownedMember, redefines ownedAttribute, ordered}
+ownedAttribute
*
Property
{subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+class
0..1
{subsets feature, subsets ownedMember, subsets redefinableElement, ordered}
+ownedOperation
*
Operation
{subsets featuringClassifier, subsets namespace}
+class
0..1
{subsets feature, subsets ownedMember}
+ownedReception
*
Reception
Only active classes may have classifier behaviors.
Only an active class may specialize an active class.

Figure 7.16 - Classes

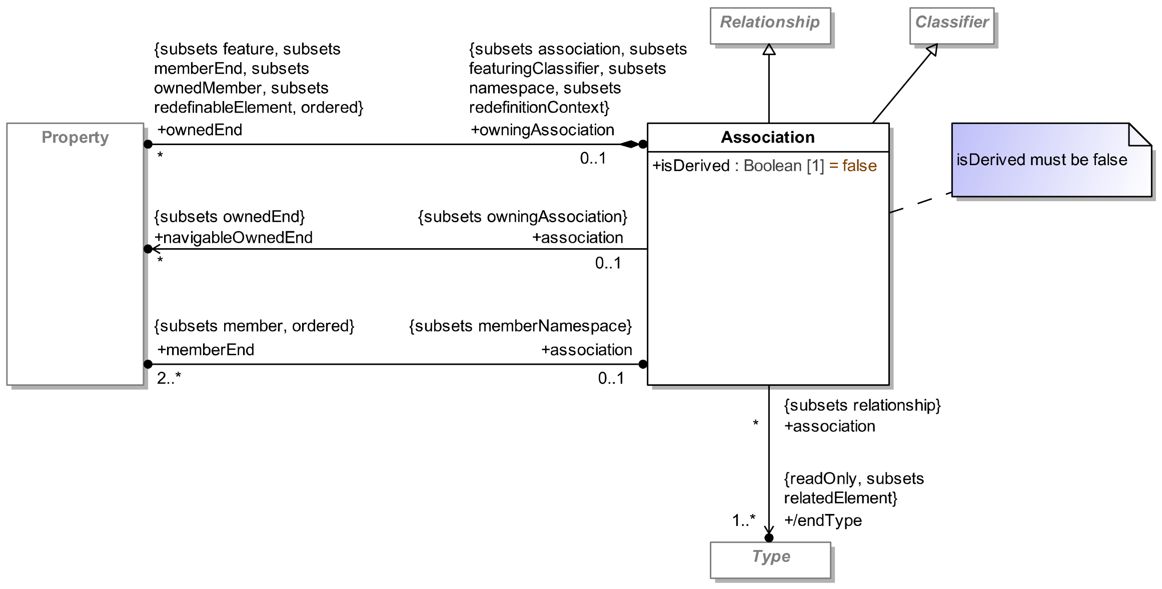

Relationship
Classifier
{subsets feature, subsets memberEnd, subsets ownedMember, subsets redefinableElement, ordered}
+ownedEnd
*
{subsets association, subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+owningAssociation
0..1
Property
Association
+isDerived : Boolean [1] = false
isDerived must be false
{subsets ownedEnd}
+navigableOwnedEnd
*
{subsets owningAssociation}
+association
0..1
{subsets member, ordered}
+memberEnd
2..*
{subsets memberNamespace}
+association
0..1
{subsets relationship}
+association
*
{readOnly, subsets relatedElement}
+/endType
1..*
Type

Figure 7.17 - Associations

7.7.2 Constraints

7.7.2.1 Association

[1] fuml_association_no_derivation
isDerived must be false

not self.isDerived

7.7.2.2 Class

[1] fuml_class_active_class_classifier_behavior
Only active classes may have classifier behaviors.

self.classifierBehavior→notEmpty() implies self.isActive

[2] fuml_class_active_class_specialization
Only an active class may specialize an active class.

self.parents()->exist(isActive) implies self.isActive

[3] fuml_class_abstract_class
Only an abstract class may have abstract behavioral features.

self.member->select(oclIsKindOf(BehavioralFeature))->exists(isAbstract) implies self.isAbstract

7.8 Packages

7.8.1 Overview

The fUML Packages package imports classes from the UML Packages package. The classes shown in Figure 7.18 are those included in the fUML Packages package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Packages (see Figure 7.18):

- PackageMerge – Package merge is excluded from fUML because it is not considered to be a runtime construct. All package merges are assumed to have been already carried out before a model is submitted for execution.
- Model – This is excluded from fUML because it is not considered to be runtime construct. For an executable model, a model package is considered equivalent to a regular package.

From Profiles (no corresponding fUML diagram):

- Profiles are excluded as being outside the scope of fUML.

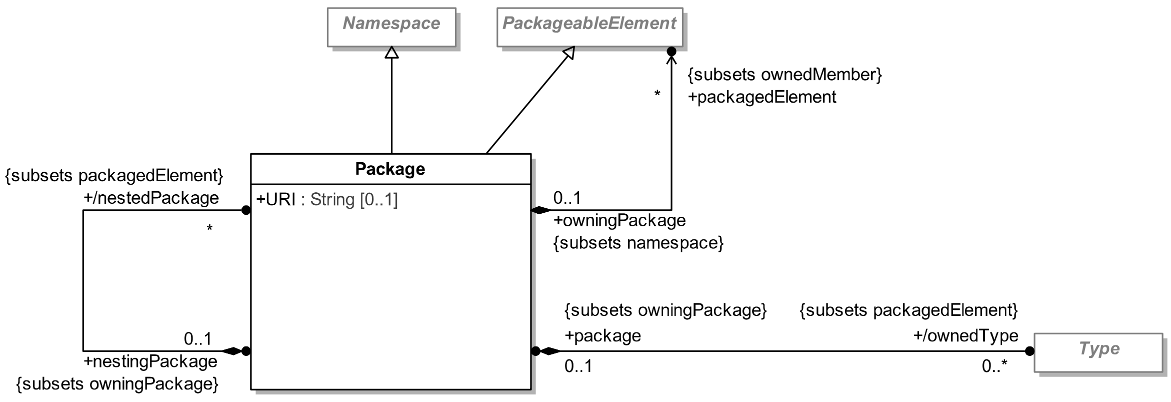


Figure 7.18 - Packages

7.8.2 Constraints

None.

7.9 Common Behavior

7.9.1 Overview

The fUML CommonBehavior package imports classes from the UML CommonBehavior package. The classes shown in Figures 7.19 and 7.20 are those included in the fUML CommonBehavior package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Common Behavior (see Figure 7.19):

- Behavior::redefinedBehavior – Behavior redefinition is excluded from fUML because opaque behaviors are only used for primitive behaviors in fUML, and the only other type of behavior provided is activities, the semantics of redefinition for which is not fully defined in UML 2.
- Behavior::precondition and Behavior::postcondition – Behavior preconditions and postconditions are excluded from fUML because constraints, in general, are excluded from fUML (see 7.3.1).

From Events (see Figure 7.20):

- TimeEvent and ChangeEvent – These events are excluded from fUML because they imply a background infrastructure, such as a model of time or a mechanism for monitoring for change. The execution semantics for this would be complicated to specify and more sophisticated than is necessary for computational completeness of the foundational subset.
- AnyReceiveEvent – Any receive events are excluded because they are largely unnecessary. Only asynchronous signal events are allowed in fUML.

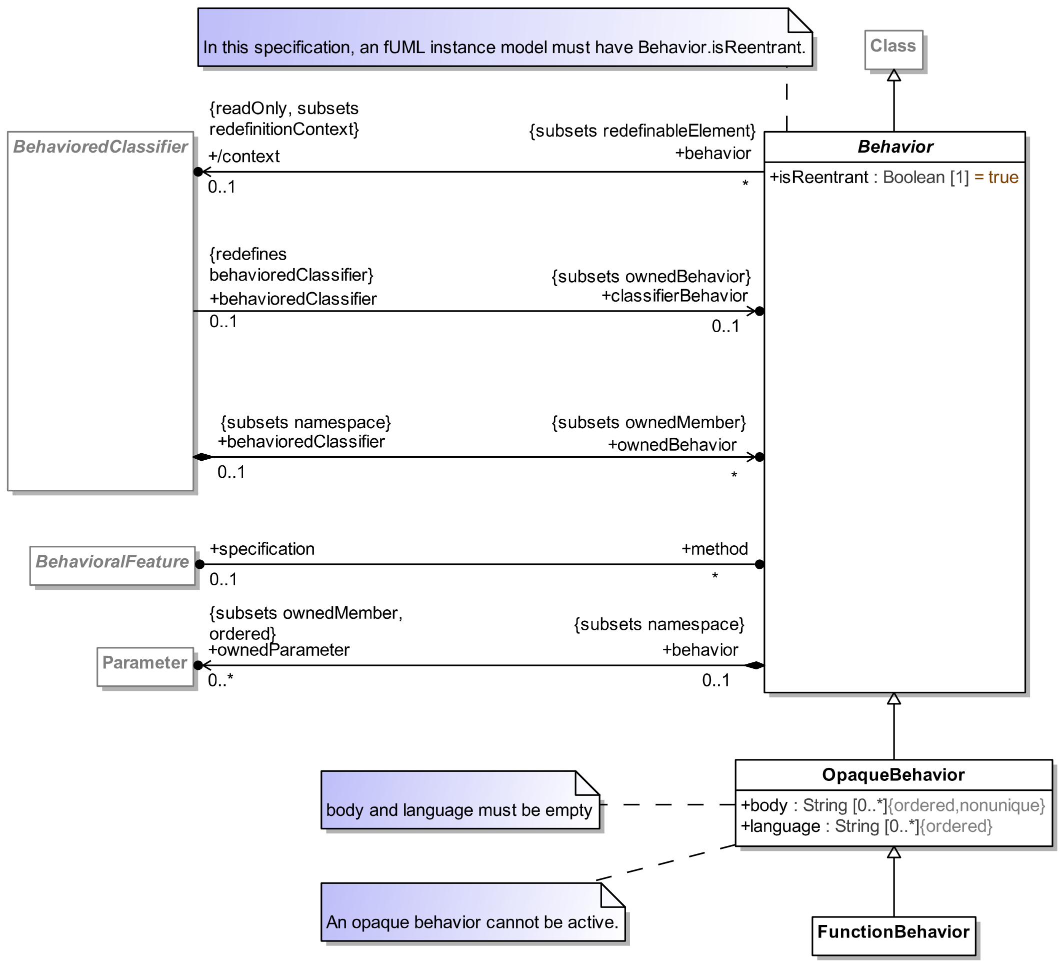

In this specification, an fUML instance model must have Behavior.isReentrant.
Class
BehavioredClassifier
{readOnly, subsets redefinitionContext}
+/context
0..1
{subsets redefinableElement}
+behavior
*
Behavior
+isReentrant : Boolean [1] = true
{redefines behavioredClassifier}
+behavioredClassifier
0..1
{subsets ownedBehavior}
+classifierBehavior
0..1
{subsets namespace}
+behavioredClassifier
0..1
{subsets ownedMember}
+ownedBehavior
*
BehavioralFeature
+specification
0..1
+method
*
Parameter
{subsets ownedMember, ordered}
+ownedParameter
0..*
{subsets namespace}
+behavior
0..1
body and language must be empty
OpaqueBehavior
+body : String [0..*]{ordered,nonunique}
+language : String [0..*]{ordered}
An opaque behavior cannot be active.
FunctionBehavior

Figure 7.19 - Behaviors

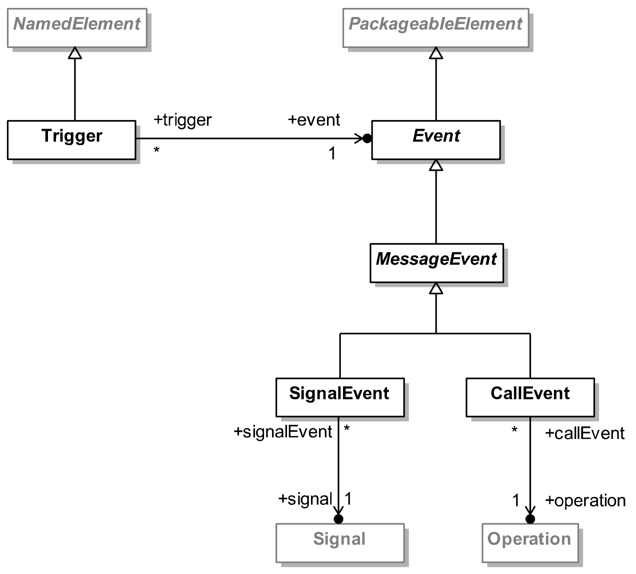


Figure 7.20 - Events

7.9.2 Constraints

7.9.2.1 Behavior

[1] fuml_behavior_reentrant
In this specification, a fUML instance model must have Behavior.isReentrant

self.isReentrant

7.9.2.2 OpaqueBehavior

[1] fuml_opaque_behavior_empty_body_and_language
body and language must be empty

self.language->isEmpty() and self.body->isEmpty()

[2] fuml_opaque_behavior_inactive
An opaque behavior cannot be active.

not self.isActive

7.10 Activities

7.10.1 Overview

The fUML Activities package imports classes from the UML Activities package. The classes shown in Figures 7.21 and 7.25 are those included in the fUML Activities package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Activities (see Figure 7.21):

- ActivityEdge::redefinedEdge – Activity edge redefinition is excluded from fUML because behavior redefinition is excluded from fUML (see 7.9).
- ActivityEdge::weight – Activity edge weights are excluded as being outside the scope of fUML.
- ActivityNode::redefinedNode – Activity node redefinition is excluded from fUML because behavior redefinition is excluded from fUML (see 7.9).
- ObjectNode::transformation and ObjectNode::selection – Transformation and selection behaviors are excluded as being outside the scope of fUML.
- Variable – Variables are excluded from fUML because the passing of data between actions can be achieved using object flows.

From Control Nodes (see Figure 7.22):

- JoinNode::joinSpec – Join specifications are excluded as being two imprecisely defined in UML to be executable.

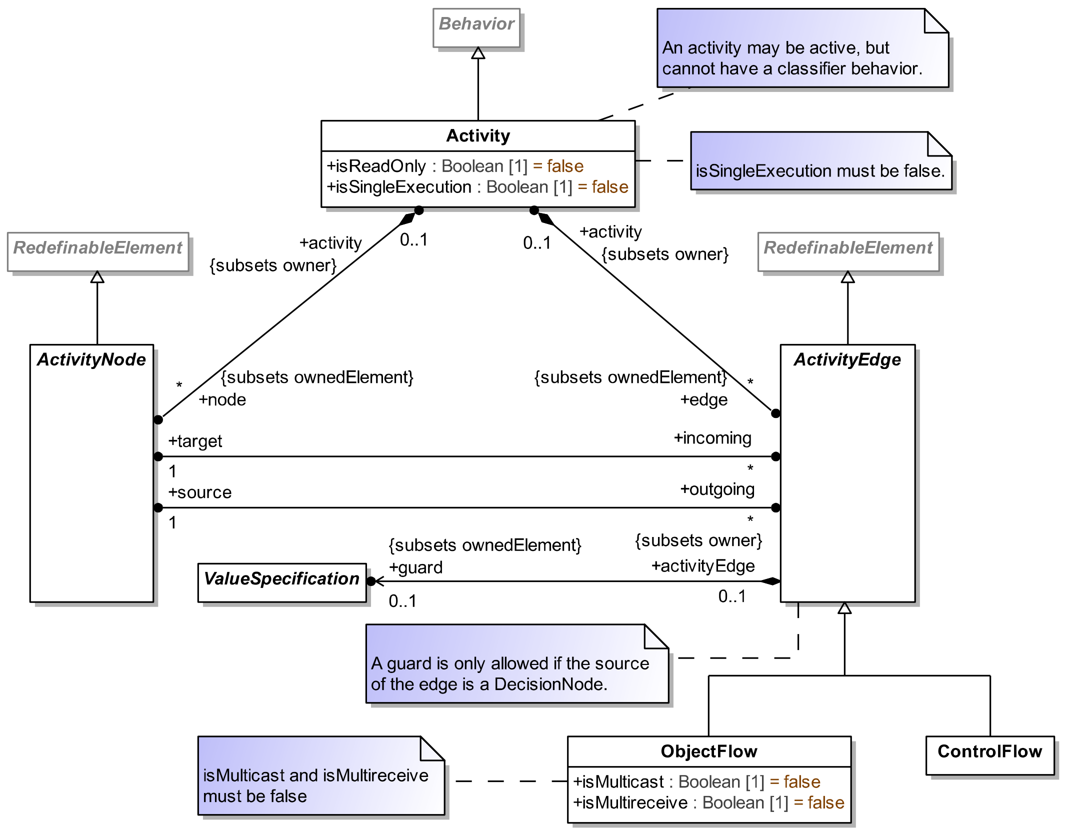
From Object Nodes (see Figure 7.23):

- ObjectNode::selection and ObjectNode::upperBound – Object node selection and upper bound are excluded as being outside the scope of fUML.
- ObjectNode::inState – Identifying a state with an object node is excluded from fUML because state machines are not included in fUML.

From Executable Nodes (see Figure 7.24): No exclusions.

From Activity Groups (see Figure 7.25):

- ActivityPartition – Activity partitions are excluded from fUML because they are a very general modeling construct in UML activities and their precise execution semantics is unclear.
- InterruptibleRegion – Interruptible regions are excluded from fUML because they are considered to be more appropriate for “higher level” process modeling and outside the scope of fUML.



Behavior
An activity may be active, but cannot have a classifier behavior.
Activity
+isReadOnly : Boolean [1] = false
+isSingleExecution : Boolean [1] = false
isSingleExecution must be false.
RedefinableElement
+activity {subsets owner}
0..1
0..1
+activity {subsets owner}
RedefinableElement
ActivityNode
ActivityEdge
* {subsets ownedElement} +node
{subsets ownedElement} +edge *
+target 1
+incoming *
+source 1
+outgoing *
{subsets ownedElement} +guard 0..1
{subsets owner} +activityEdge 0..1
ValueSpecification
A guard is only allowed if the source of the edge is a DecisionNode.
ObjectFlow
+isMulticast : Boolean [1] = false
+isMultireceive : Boolean [1] = false
ControlFlow
isMulticast and isMultireceive must be false

Figure 7.21 - Activities

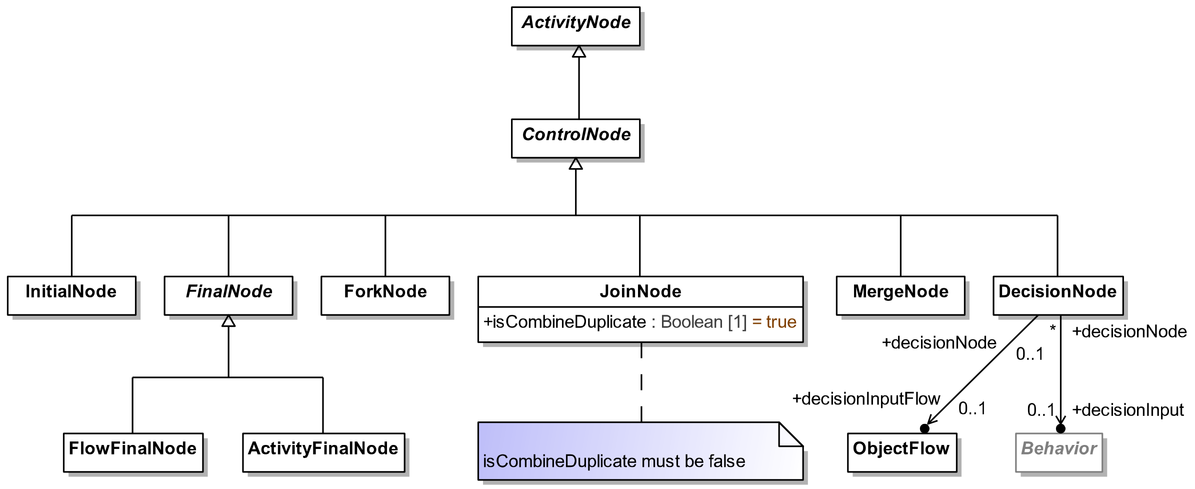


Figure 7.22 - Control Nodes

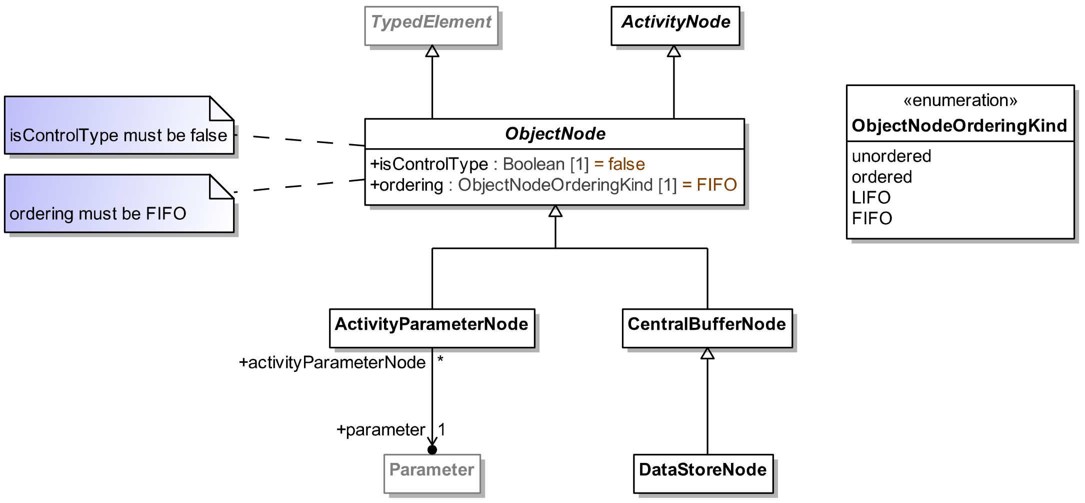


Figure 7.23 - Object Nodes

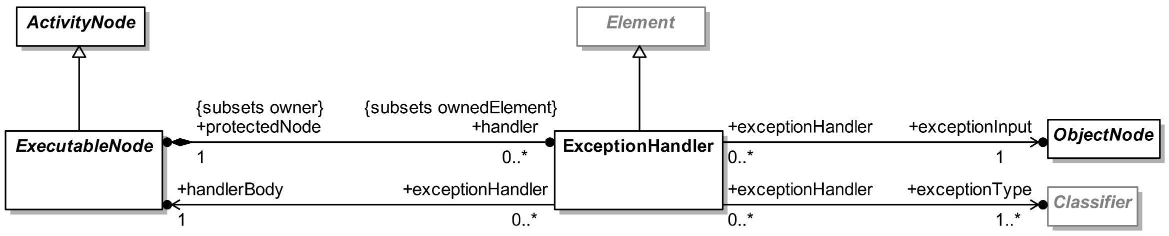


Figure 7.24 - Executable Nodes

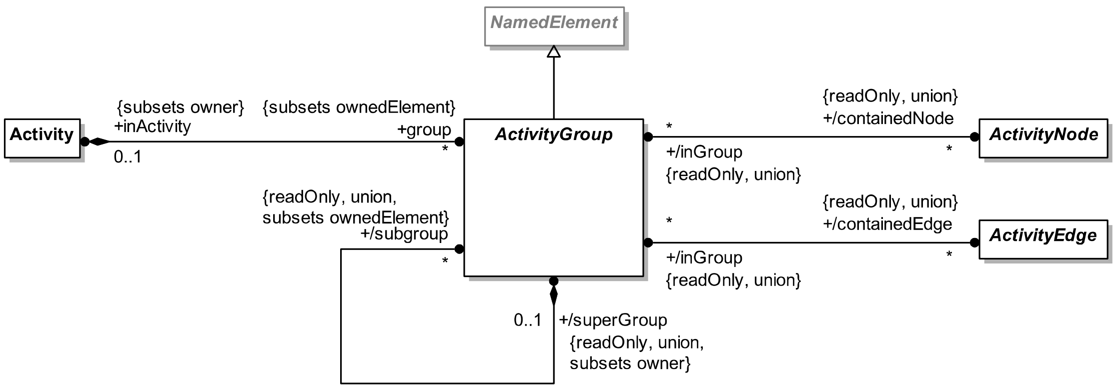


Figure 7.25 - Activity Groups

7.10.2 Constraints

7.10.2.1 Activity

[1] fuml_activity_no_classifier_behavior
An activity may be active, but cannot have a classifier behavior.

```
self.classifierBehavior->isEmpty()
```

[2] fuml_activity_not_single_execution
isSingleExecution must be false.

```
not self.isExecution
```

7.10.2.2 ActivityEdge

[1] fuml_activity_edge_allowed_guards
A guard is only allowed if the source of the edge is a DecisionNode.

```
self.guard->notEmpty() implies self.source.oclIsKindOf(DecisionNode)
```

7.10.2.3 JoinNode

[1] fuml_join_node_not_combine_duplicate
isCombineDuplicate must be false

```
not self.isCombineDuplicate
```

7.10.2.4 ObjectFlow

[1] fuml_object_flow_not_multi
isMulticast and isMultireceive must be false

```
not self.isMulticast and not self.isMultireceive
```

7.10.2.5 ObjectNode

[1] fuml_object_node_fifo_ordering
ordering must be FIFO

```
self.ordering = ObjectNodeOrderingKind::FIFO
```

[2] fuml_object_node_not_control_type
isControlType must be false

```
not self.isControlType
```

7.11 Actions

7.11.1 Overview

The fUML Actions package imports classes from the UML Actions package. The classes shown in Figures 7.26 and 7.35 are those included in the fUML Actions package. The diagrams correspond to similar diagrams in the UML 2 Specification. The following classes and features have been excluded from the fUML subset and are, therefore, not shown on the fUML abstract syntax diagrams.

From Actions (see Figure 7.26):

- Action::localPrecondition and Action::localPostcondition – Local preconditions and postconditions for actions are not supported in fUML because constraints are not supported in fUML (see 7.3).
- ActionInputPin – Action input pins are excluded from fUML because they are redundant with using an object flow to connect the output pin of the action to a regular input pin.
- OpaqueAction – Opaque actions are excluded from fUML since, being opaque, they cannot be executed.
- ValuePin – Value pins are excluded from fUML because they are redundant with using value specifications to specify values.

From Invocation Actions (see Figure 7.27):

- InvocationAction::onPort – Identification of a port for an invocation action is excluded from fUML because ports are excluded from fUML.
- BroadcastSignalAction and SendObjectAction – The sole mechanism for asynchronous invocation in fUML is via send signal action. This can be used to achieve the effect of broadcasting and sending objects.

From Object Actions (see Figure 7.28): No exclusions.

From Link End Data (see Figure 7.29):

- QualifierValue – Qualifier values are excluded from fUML because association qualifiers are excluded from fUML (see 7.7).

From Link Actions (see Figure 7.30): No exclusions.

From Link Object Actions (no fUML diagram):

- ReadLinkObjectEndAction, ReadLinkObjectEndQualifierAction and CreateLinkObjectAction – These actions are excluded from fUML because association classes are excluded from fUML (see 7.7).

From Structural Feature Actions (see Figure 7.31): No exclusions.

From Variable Actions (no fUML diagram):

- ReadVariableAction, WriteVariableAction (and its subclasses) and ClearVariableAction – These actions are excluded from fUML because variables are excluded from fUML (see 7.10).

From Structured Actions (see Figure 7.33):

- StructuredActivityNode::variables – Variables are excluded from fUML because the passing of data between actions can be achieved using object flows (see also 7.10).
- SequenceNode – Sequence nodes are excluded from fUML because the sequencing of actions can be expressed using control flows.

From Expansion Regions (see Figure 7.34): No exclusions.

From Other Actions (see Figure 7.35): No exclusions.

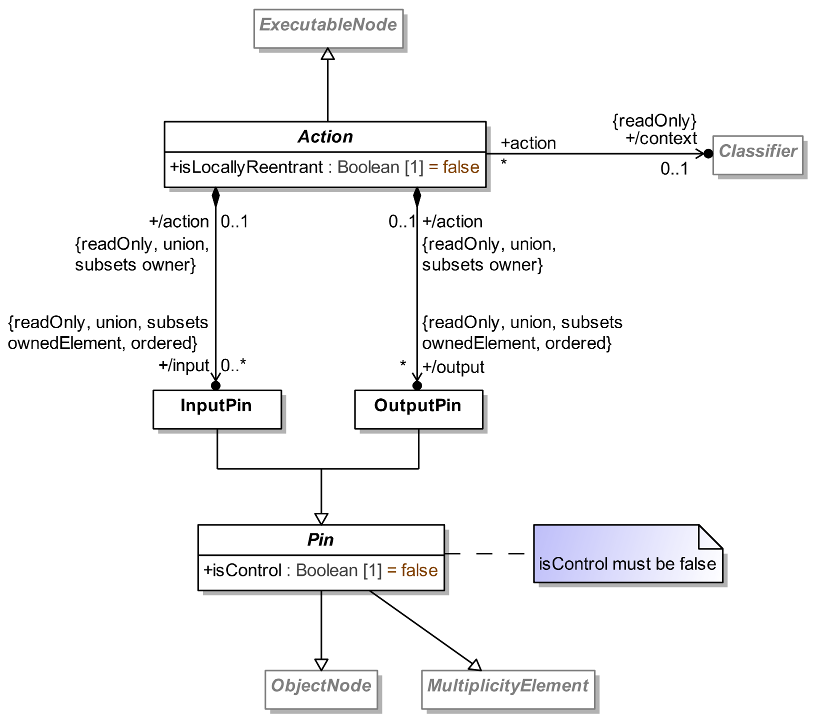

ExecutableNode
Action
+isLocallyReentrant : Boolean [1] = false
+action
*
{readOnly}
+/context
0..1
Classifier
+/action
0..1
{readOnly, union, subsets owner}
{readOnly, union, subsets ownedElement, ordered}
+/input
0..*
0..1
+/action
{readOnly, union, subsets owner}
{readOnly, union, subsets ownedElement, ordered}
*
+/output
InputPin
OutputPin
Pin
+isControl : Boolean [1] = false
isControl must be false
ObjectNode
MultiplicityElement

Figure 7.26 - Actions

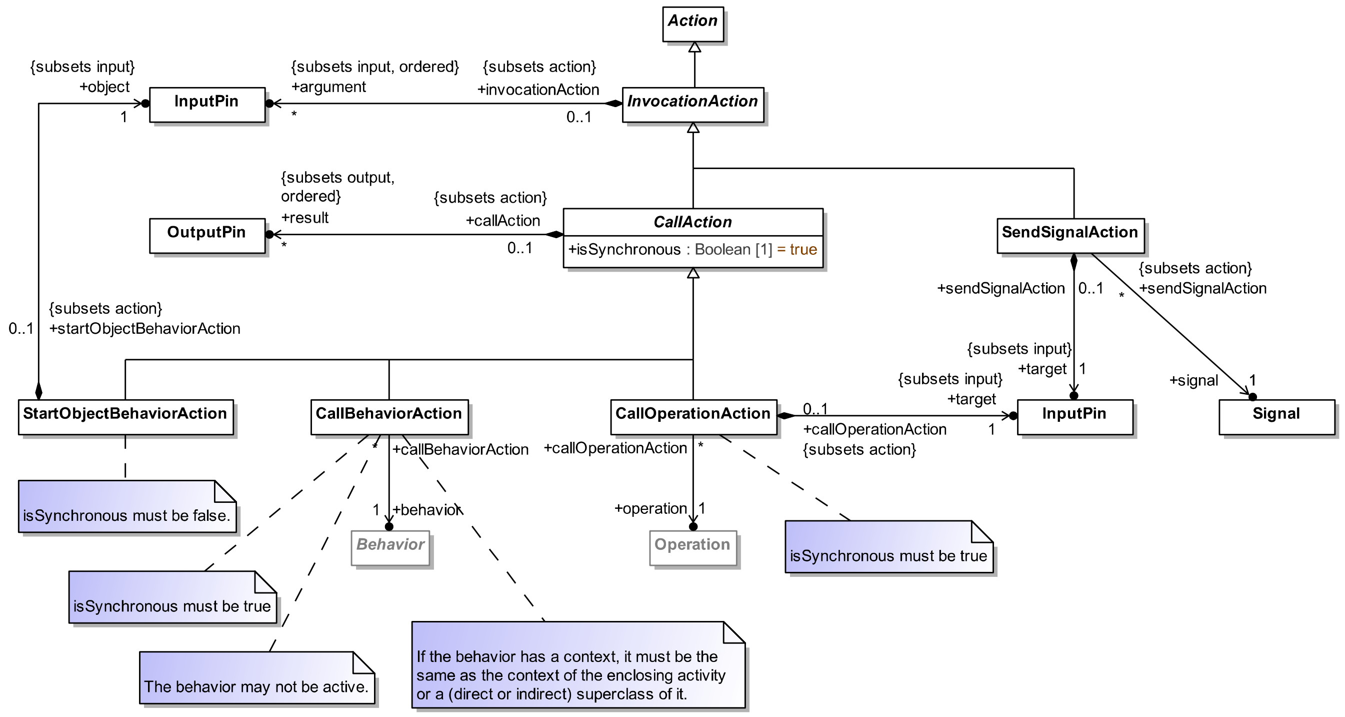
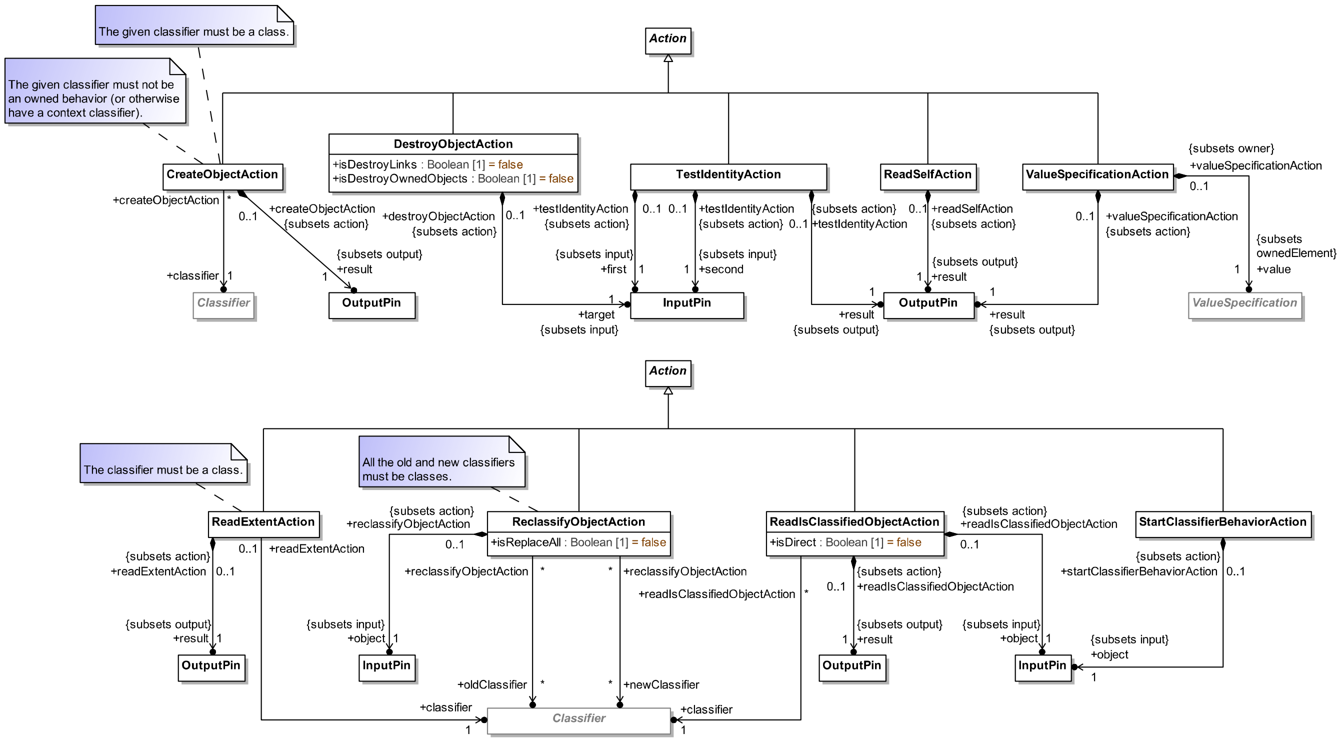


Figure 7.27 - Invocation Actions



The given classifier must be a class.
The given classifier must not be an owned behavior (or otherwise have a context classifier).
Action
CreateObjectAction
DestroyObjectAction
+isDestroyLinks : Boolean [1] = false
+isDestroyOwnedObjects : Boolean [1] = false
TestIdentityAction
ReadSelfAction
ValueSpecificationAction
Classifier
OutputPin
InputPin
ValueSpecification
+createObjectAction
+classifier
{subsets output}
+result
+destroyObjectAction
{subsets action}
+target
{subsets input}
+testIdentityAction
+first
+second
+readSelfAction
+valueSpecificationAction
{subsets owner}
{subsets ownedElement}
+value
All the old and new classifiers must be classes.
The classifier must be a class.
ReadExtentAction
ReclassifyObjectAction
+isReplaceAll : Boolean [1] = false
ReadIsClassifiedObjectAction
+isDirect : Boolean [1] = false
StartClassifierBehaviorAction
+readExtentAction
+reclassifyObjectAction
+readIsClassifiedObjectAction
+startClassifierBehaviorAction
+object
+oldClassifier
+newClassifier

Figure 7.28 - Object Actions

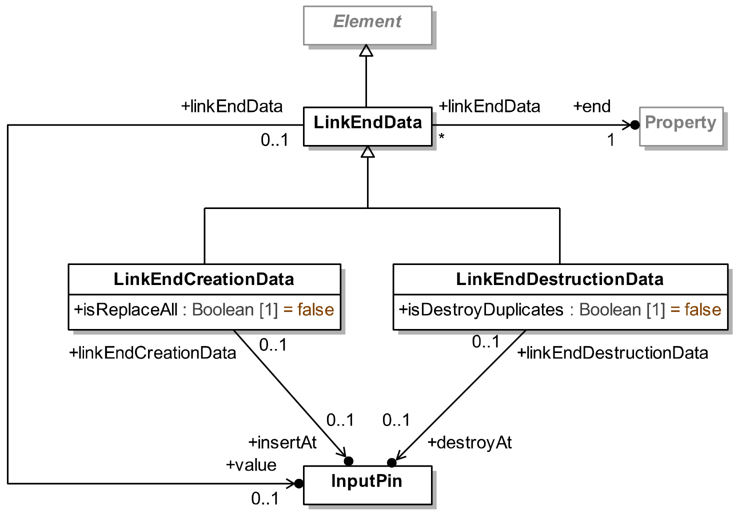


Figure 7.29 - Link End Data

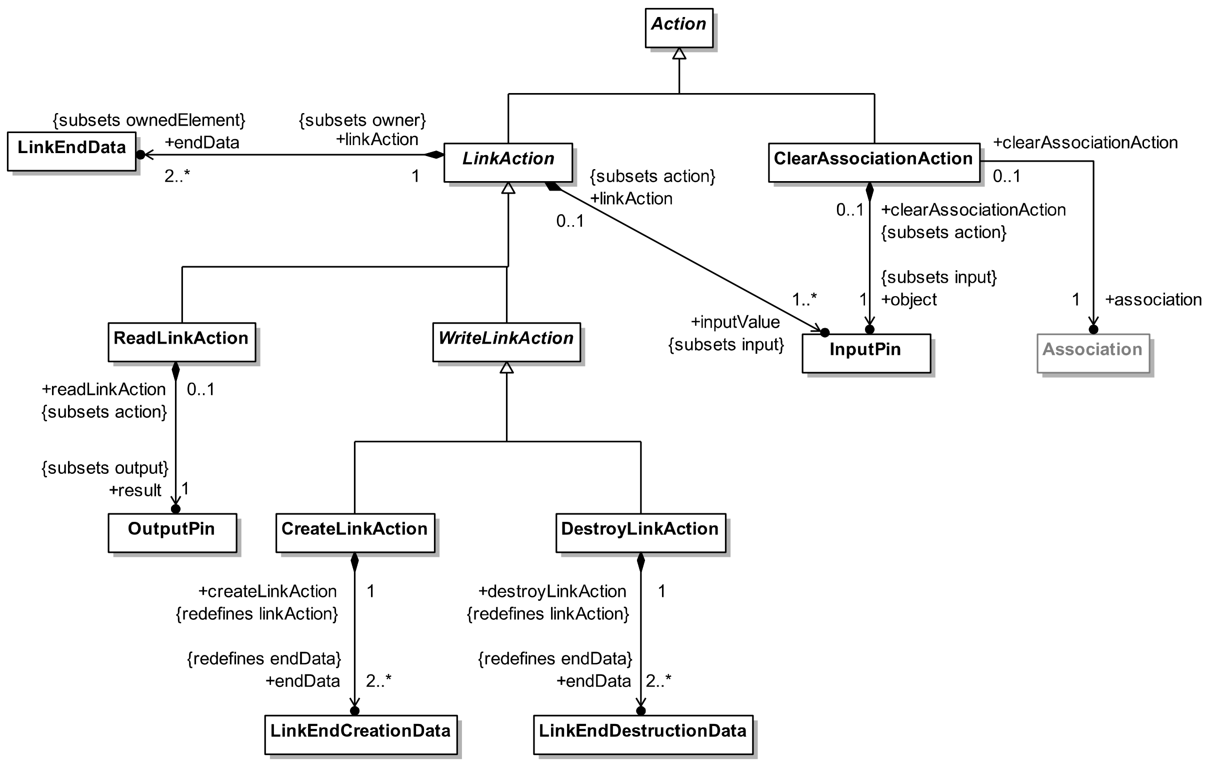


Figure 7.30 - Link Actions

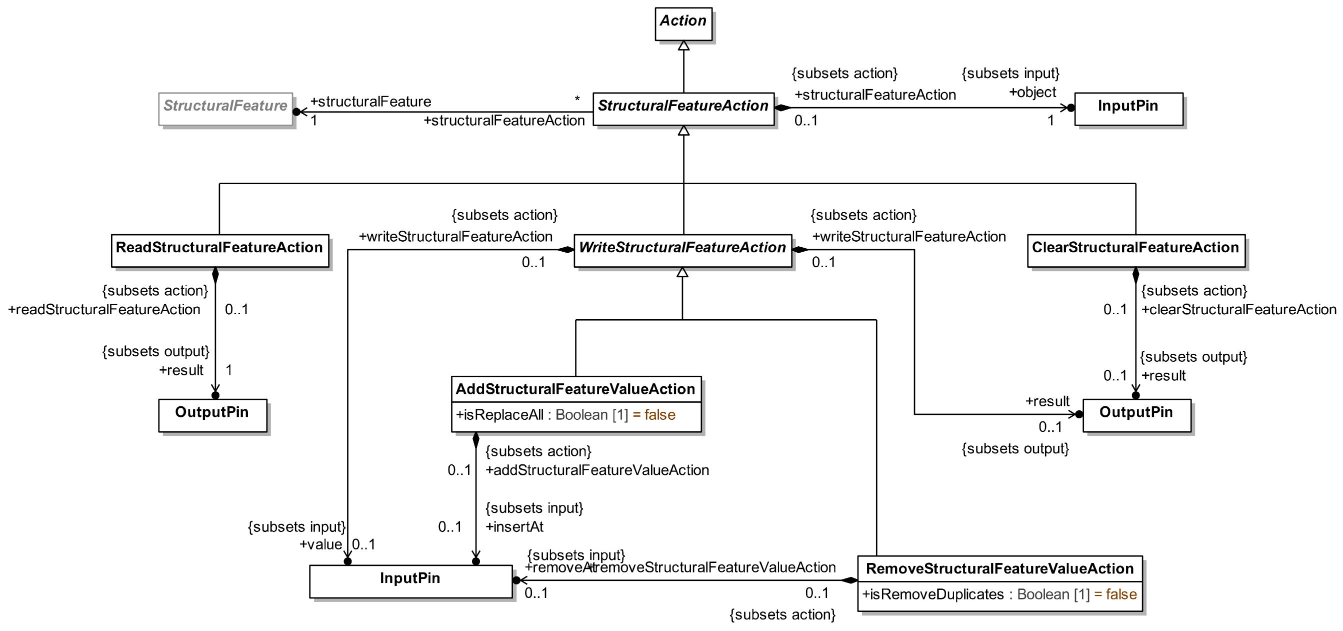


Figure 7.31 - Structural Feature Actions

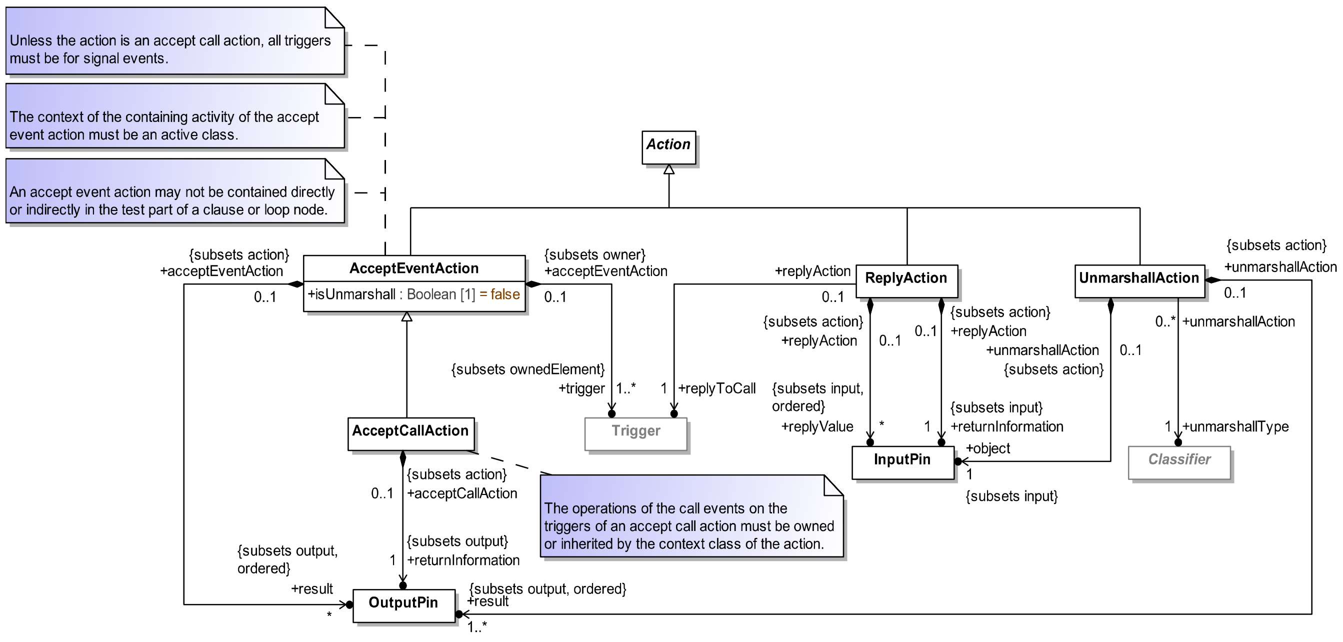


Figure 7.32 - Accept Event Actions

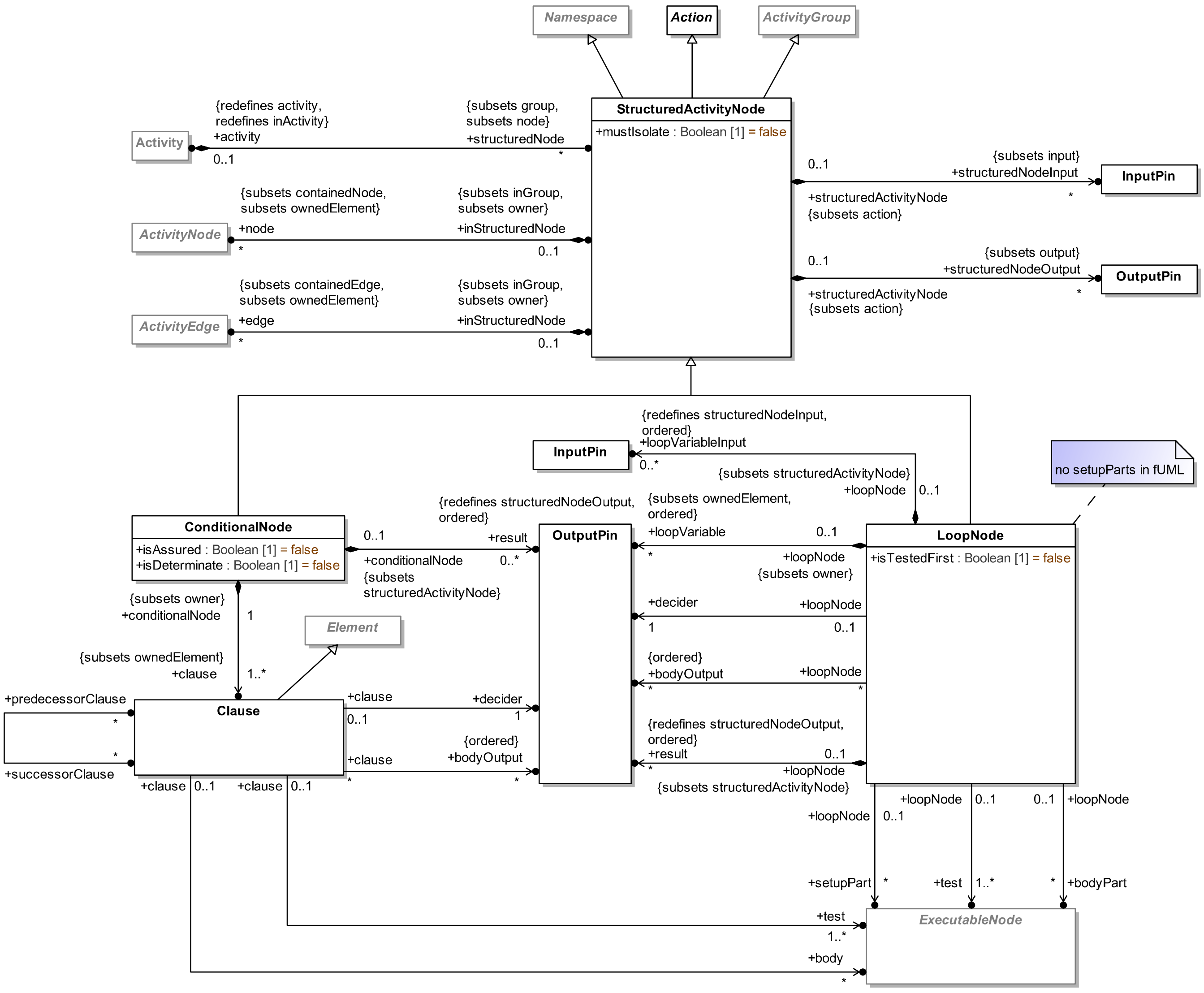


Figure 7.33 - Structured Actions

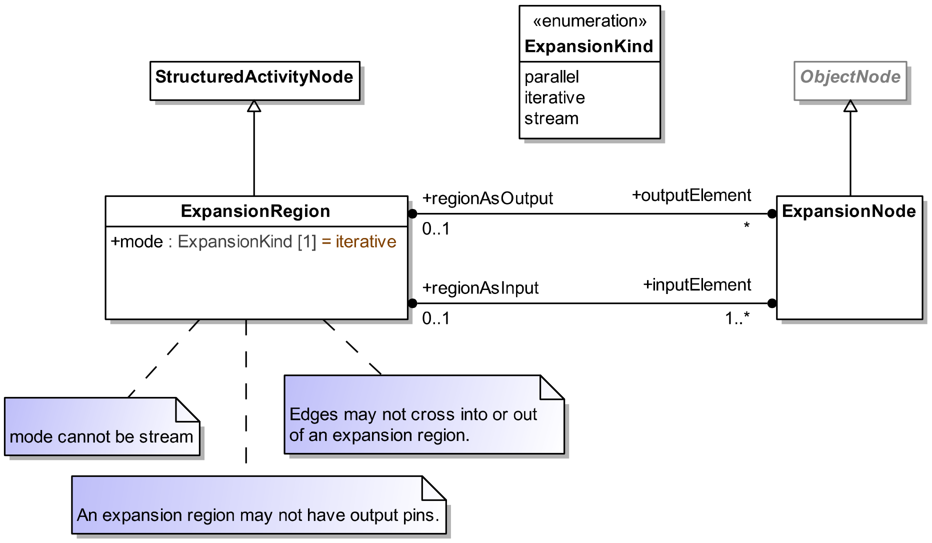


Figure 7.34 - Expansion Regions

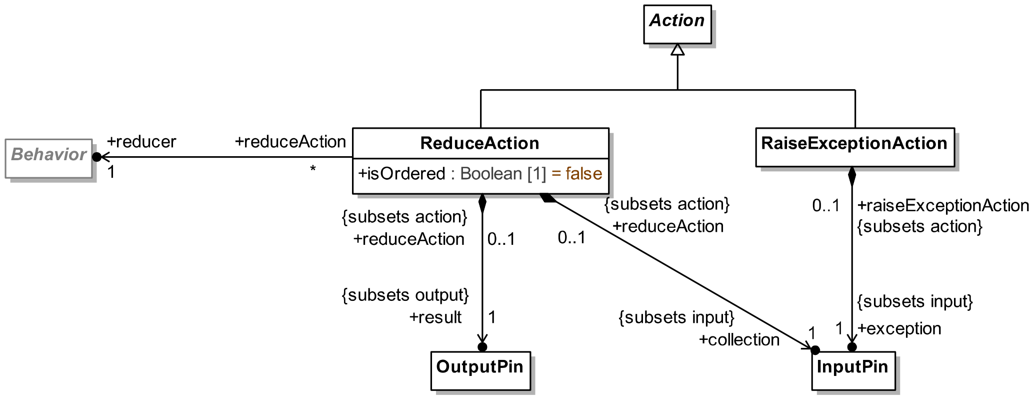


Figure 7.35 - Other Actions

7.11.2 Constraints

7.11.2.1 AcceptCallAction

[1] fuml_accept_call_action_call_event_operations
The operations of the call events on the triggers of an accept call action must be owned or inherited by the context class of the action.

```
let cls: Class = self.context.oclAsType(Class) in
let classes:Bag(Class) = cls.allParents()->select(oclIsKindOf(Class))->collect(oclAsType(Class))->union(cls->asBag()) in
classes.ownedOperation→includesAll(self.trigger.event→collect(oclAsType(CallEvent)).operation)
```

7.11.2.2 AcceptEventAction

[1] fuml_accept_event_action_active_context
The context of the containing activity of the accept event action must be an active class.

```
self.context.oclAsType(Class).isActive
```

[2] fuml_accept_event_no_accept_event_action_in_tests
An accept event action may not be contained directly or indirectly in the test part of a clause or loop node.

```
self->closure(inStructuredNode.oclAsType(ActivityNode))->forAll(n |
        let s : StructuredActivityNode = n.inStructuredNode in
        s->notEmpty() implies
             (s.oclIsTypeOf(ConditionalNode) implies s.oclAsType(ConditionalNode).clause.test->
                  excludes(n.oclAsType(ExecutableNode)) and
             s.oclIsTypeOf(LoopNode) implies s.oclAsType(LoopNode).test->excludes(n.oclAsType(ExecutableNode))))
```

[3] fuml_accept_event_only_signal_event_triggers
Unless the action is an accept call action, all triggers must be for signal events.

```
not self.oclIsKindOf(AcceptCallAction) implies
self.trigger.event->forAll(oclIsKindOf(SignalEvent))
```

7.11.2.3 CallBehaviorAction

[1] fuml_call_behavior_action_inactive_behavior
The behavior may not be active.

```
not self.behavior.isActive
```

[2] fuml_call_behavior_action_is_synchronous
isSynchronous must be true

```
self.isSynchronous
```

[3] fuml_call_behavior_action_proper_context
If the behavior has a context, it must be the same as the context of the enclosing activity or a (direct or indirect) superclass of it.

```
self.behavior.context->notEmpty() implies
  self.context->union(self.context.allParents())->includes(self.behavior.context)
```

7.11.2.4 CallOperationAction

[1] fuml_call_operation_action_is_synchronous
isSynchronous must be true

```
self.isSynchronous
```

7.11.2.5 CreateObjectAction

[1] fuml_create_object_action_is_class
The given classifier must be a class.

```
self.classifier.oclIsKindOf(Class)
```

[2] fuml_create_object_action_no_owned_behavior
The given classifier must not be an owned behavior (or otherwise have a context classifier).

```
self.classifier.oclIsKindOf(Behavior) implies self.classifier.oclAsType(Behavior).context = null
```

7.11.2.6 ExpansionNode

[1] fuml_expansion_node_mode_cannot_be_stream
mode cannot be stream

```
self.mode <> ExpansionKind::stream
```

[2] fuml_expansion_node_no_crossing_edges
Edges may not cross into or out of an expansion region.

```
self.edge->forAll(self.node->includes(source) and self.node→includes(target))
```

[3] fuml_expansion_node_no_output_pins
An expansion region may not have output pins.

```
self.output->isEmpty()
```

7.11.2.7 LoopNode

[1] fuml_loop_node_no_setup_part
no setupParts in fUML

```
self.setupPart->isEmpty()
```

7.11.2.8 Pin

[1] fuml_pin_not_control
isControl must be false

```
not self.isControl
```

7.11.2.9 ReadExtentAction

[1] fuml_read_extent_action_is_class
The classifier must be a class.

```
self.classifier.oclIsKindOf(Class)
```

7.11.2.10 ReclassifyObjectAction

[1] fuml_reclassify_object_action_old_new_classes
All the old and new classifiers must be classes.

```
self.oldClassifier->forAll(oclIsKindOf(Class)) and self.newClassifier→forAll(oclIsKindOf(Class))
```

7.11.2.11 StartObjectBehaviorAction

[1] fuml_start_object_behavior_action_is_asynchronous
isSynchronous must be false.

```
not self.isSynchronous
```

8 Execution Model

8.1 Overview

This clause describes the execution model for fUML. The execution model is itself a model, written in fUML, that specifies how fUML models are to be executed. This circularity is broken by the separate specification of a base semantics for the subset of fUML actually used in the execution model (see Clause 10).

Static Semantics and Well Formedness

It is important to distinguish execution semantics from what is sometimes called "static semantics," a term that comes from programming language compiler theory.

Typically, the syntax of a programming language is defined using a context-free grammar (e.g., using Backus-Naur Form productions). However, there are also typically aspects of the language that are context-sensitive, but can still be checked statically by the compiler. The most common example is static type checking, which requires matching expression types to be declared variable types. The checking of such context-sensitive constraints is known as "static semantics."

For UML, the abstract syntax is defined as a MOF metamodel. The UML specification also defines additional constraints that the metamodel representation of a valid UML model is required to meet. These constraints are the equivalent of the static semantics of UML.

However, since these constraints can all be checked statically, they are not part of the execution semantics of UML. Indeed, any model that violates one or more of these additional constraints is not actually well formed. Such an ill-formed model cannot really be assigned any meaning at all.

In this specification, static semantics are not considered to be part of the execution semantics to be specified. That is, any well-formed model is already presumed to have met all the constraints imposed on the abstract syntax as defined in the UML Specification. Semantic meaning will only be defined for models that are well formed in this sense.

Conventions on Derivation and Redefinition

In a number of cases in the UML abstract syntax metamodel, constraints express requirements for derived properties (including the implicit constraints involved in derived unions and subsetting). The values of such properties may be completely determined from the values of other, non-derived properties using the defining constraints. Thus, for example, the values of these properties do not need to be included in the interchange representation of the model.

On the other hand, the UML 2 Specification allows a derived property to be read using a read structural feature action, just like any other property. In principle, it should be possible to dynamically compute the value of the derived property in order to read it. However, the fUML subset does not include constraints (see 7.3.1 for the rationale for this exclusion) and, therefore, the defining constraints for derived properties are not available in an executing fUML model.

As a result, this specification adopts the convention that, when an object is instantiated, explicit values are provided for all derived properties and that these values are consistent with the defining constraints for the derivation. In the context of the abstract syntax metamodel, this means that all the implicit and explicit derivation constraints are treated as part of the conditions for a well-formed model. Consistent with the discussion of well-formedness above, the execution model therefore assumes that the abstract syntax representation of a model being executed has valid values set for all derived properties that may be read just like other properties (and that all derived properties keep the same value throughout an execution). That is, the distinction between derived and non-derived properties essentially disappears at runtime, so far as the execution model is concerned (since the execution model does not change the value of any properties in the abstract syntax representation of an input model).

For example, the UML abstract syntax metamodel defines the ownedAttribute property of Class to subset the derived union Namespace::ownedMember, which, in turn, subsets both Namespace::member and Element::ownedElement. The fUML execution model assumes that, in the abstract syntax representation of a well-formed model, every ownedAttribute of the representation of a class will also be explicitly included in the collection of values of the inherited ownedMember, member and ownedElement properties for that class.

Similarly, an object is considered to have values set for both any redefined property and the property redefinition of it. In this case, the implicit constraint is that the values must be the same, whether accessed via the redefined property or via the redefining property. However, the redefining property may also impose additional constraints (such as a narrowing of the allowed multiplicity, for example) that then effectively also apply to the value of the redefined property.

Note: A conforming execution tool is not necessarily required to handle the derived and redefined properties of the UML abstract syntax metamodel in this way. This is simply the convention for the execution model, which is written within the constraints of the fUML subset.

Behavioral Semantics

The execution model is a formal, operational specification of the execution semantics of fUML. That is, it defines the operational procedure for the dynamic changes required during the execution of a fUML model. This is in contrast to the declarative approach used for the base semantics (see Clause 10).

The execution model is itself an executable, object-oriented, fUML model of a fUML execution engine. To specify the behavioral semantics of fUML completely, the execution model must fully define its own behavior-that is, it must fully specify every operation method and classifier behavior in it. Since the only kind of user-defined behavior supported in fUML is the activity, each behavior in the execution model must be modeled as an activity.

Currently, the only UML notation provided for activity modeling is the graphical activity diagram. It would thus be possible to represent each of the activities in the execution model using such a diagram. For example, Figure 8.14 gives a sample activity diagram for just a part of the method specified for the execute operation of the ActivityExecution class in the execution model. Unfortunately, for significant activities, these diagrams quickly become large, intractable to draw and hard to comprehend.

Instead of using such cumbersome graphical notation, and rather than defining from scratch a new, non-normative textual notation for activities, most activities in this specification are written as equivalent code in the Java programming language. Informally, these code snippets can actually be understood as executable Java code, and the standard Java semantics for this code is consistent with the behavior to be specified for the activity. For example, Figure 8.15 shows the Java code equivalent to the activity model in Figure 8.14.

Formally, however, any Java code should be understood as just a surface notation for the true, underlying UML activity model. That is, the code in Figure 8.15 should be thought of as just another representation of the model given in Figure 8.14. Annex A provides the normative mapping from this Java surface notation to UML activity models, for the purposes of the fUML specification. The formal semantics of the constructs used in activity models mapped from the Java surface notation is then given by the base semantics in Clause 10.

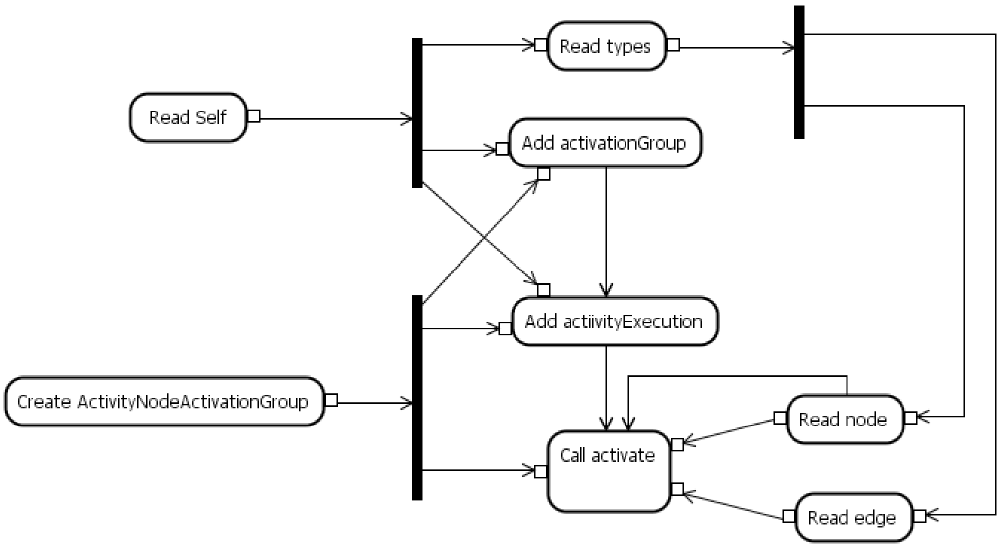


Figure 8.1 - Partial Activity Model for the ActivityExecution::execute Operation

```
Activity activity = (fUML.Syntax.Activity) (this.types.getValue(0));
ActivityNodeActivationGroup group = new ActivityNodeActivationGroup();
this.activationGroup = group;
group.activityExecution = this;
```

Figure 8.2 - Java Surface Representation of the Activity Model in Figure 8.1

8.2 Semantics Packages

The remainder of this clause is organized according to the packaging structure of the execution model. The packaging of the execution model parallels that of the fUML abstract syntax (see Clause 7), except that the execution model does not include CommonStructure and Package packages, because these either contain abstract syntax elements without execution semantics or whose general execution semantics are accounted for in the execution model Value package. In addition, the execution model includes the Loci package, which contains elements of the execution model that do not directly correspond to syntactic elements of fUML. Rather, the elements in this package provide a model of an executor for well-formed fUML models, which can be considered to be the abstract specification for actual fUML execution engines.

Figure 8.3 shows the relationships of the fUML semantics packages with each other and the corresponding syntax packages. Subclause 8.3 describes the Loci package, which contains the Locus, Executor, and ExecutionFactory classes that model a fUML execution engine and its environment. Subclauses 8.4 to 8.7 cover the Values, Classification, SimpleClassifiers and StructuredClassifiers packages, which together define the *structural semantics* of fUML. Subclauses 8.8 to 8.10 cover the Common Behavior, Activities and Actions packages, which together define the *behavioral semantics* of fUML.

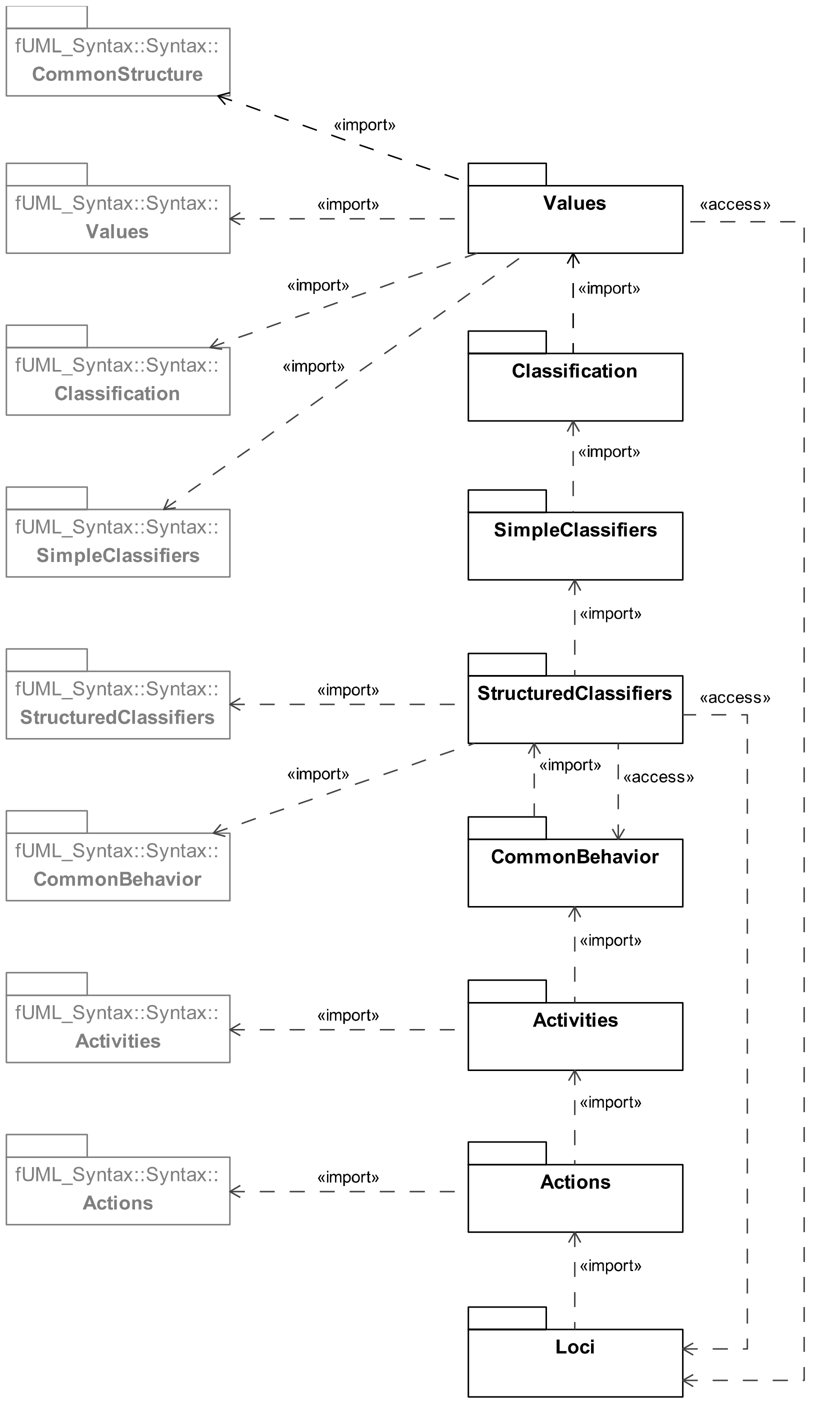


Figure 8.3 - fUML Semantics Package

Throughout the following subclauses, the terminology of semantic interpretation introduced in Clause 6 will be freely used to relate the operational semantic specification provided by the execution model to the general semantics approach used in this specification.

8.3 Loci

8.3.1 Overview

The Loci package includes the model of the key concepts of an execution *locus* and of the *executor* that provides the abstract external interface of the execution model.

The Executor and the Execution Locus

The Executor class provides the root abstraction for executing a fUML model. As shown in Figure 8.4, it provides three operations:

- Evaluate – Evaluate a value specification, returning the specified value.
- Execute – Synchronously execute a behavior, given values for its input parameters and returning values for its output behaviors.
- Start – Asynchronously start the execution of a stand-alone or classifier behavior, returning a reference to the instance of the executing behavior or of the behaviored classifier.

Every execution takes place at a specific locus. A locus is an abstraction of a physical or virtual computer capable of executing fUML models. It is a place at which extensional values (objects or links) can exist. The extent of a class or association is the set of objects or links of that type that exist at a certain locus. Note that this implies that an individual object is restricted to a single locus; i.e., it cannot span multiple loci (see 8.7.1 for further discussion of extensional values.)

All objects and links created during an execution are created at the locus of that execution. And, unless an object or link is explicitly destroyed, it will persist at the locus even after the execution has completed. This means that objects and links may already exist at a locus before a specific behavior execution begins, providing part of the environment in which the execution takes place. (The concept of an execution environment is discussed further at the end of this subclause.)

Indeed, an execution locus may provide a set of pre-existing objects as part of the environment of all behavior executions at that locus, as a means of providing external services to those executions. Given that the appropriate class is known, such service objects may be discovered using the read extent action (this is the mechanism used for accessing input/output services, for example – see 9.5). More sophisticated discovery services may also be provided but are not defined in this specification.

While the execution of any one behavior takes place at a single locus, an execution at one locus may invoke a behavior that executes at another locus. To do this, an execution must be able to instantiate, or otherwise obtain a reference to, an object on the remote locus on which the behavior is to be invoked (or which itself is a behavior instance). However, no normative mechanism is provided within fUML for an execution on one locus to obtain references to objects on another locus. Conformant execution tool implementations may optionally provide a service to discover objects on remote services or to allow references to be passed between loci using input/output channels (see 9.5). (With such extensions it should be possible to support the execution of models that span multiple loci.)

Visitor Classes and the Execution Factory

The model for evaluation and execution is based on the Visitor pattern. This pattern is used to add behavior to an already existing class hierarchy. In the case of the execution model, the existing class hierarchy is that of the fUML subset of the UML Abstract Syntax (see Clause 7). The intent of the execution model is to provide a specification for the execution of models represented in terms of instances of abstract syntax metaclasses, without making any change to those metaclasses as they are given in the UML Superstructure specification.

Using the Visitor Pattern, each abstract syntax metaclass for which behavior is to be added has a corresponding visitor class in the Execution Model. This visitor class has a unidirectional association to the corresponding abstract syntax metaclass and operations that effectively provide the behavioral specification of the semantics of model elements represented by that metaclass. All visitor classes in the execution model are descended, directly or indirectly from the root SemanticVisitor class (see Figure 8.4).

There are three types of visitor classes in the Execution Model. Two of them, evaluations and executions, are used by the Executor.

- Evaluations – An evaluation visitor is used to evaluate a specific kind of value specification; that is, to return an instance of the value denoted by the value specification. There is an evaluation visitor class corresponding to each concrete subclass of ValueSpecification included in the fUML subset (see 8.4). The name of the visitor class is the same as the name of the corresponding abstract syntax metaclass with the word "Evaluation" appended. For example, the evaluation visitor class for the abstract syntax metaclass LiteralString is called LiteralStringEvaluation. (See 8.4.1 for further discussion of evaluation classes.)
- Executions – An execution visitor is used to execute a specific kind of behavior. There is an execution visitor class corresponding to each concrete subclass of Behavior included in the fUML subset (see 8.8 and 8.9). The name of the visitor class is the same as the name of the corresponding abstract syntax metaclass with the word "Execution" appended. The primary kind of UML behavior included in fUML is the activity with a corresponding visitor class called ActivityExecution. There are also OpaqueBehaviorExecution and FunctionBehaviorExecution visitor classes corresponding to OpaqueBehavior and FunctionBehavior. (See 8.8 for a general discussion of execution classes and 8.9 for specific discussion of activity execution.)

The behavior of the Executor evaluate and execute operations is to create an instance of the corresponding evaluation or execution visitor class and then use that visitor instance to carry out the required evaluation or execution. To create a corresponding visitor instance, the Executor uses an instance of the ExecutionFactory class located at the execution locus (see Figure 8.4). The ExecutionFactory class provides createEvaluation and createExecution operations that take, respectively, value specification and behavior abstract syntax instances and return, respectively, instances of the evaluation or execution class corresponding to the concrete class of the input abstract syntax object.

The third type of visitor class is an activation. An activation visitor is used to model the semantics of a specific kind of activity node within the execution of a containing activity. Such activation instances are created as part of the construction of the execution object for an activity. Therefore, they are further described in 8.9 as part of the discussion of activity execution.

All three types of visitor classes are ultimately instantiated using the instantiateVisitor operation of the ExecutionFactory class (see Figure 8.5). The ExecutionFactory class also has three subclasses: ExecutionFactoryL1, ExecutionFactoryL2 and ExecutionFactoryL3. These are provided solely for backward compatibility with previous versions of fUML and provide no additional functionality beyond that of the base ExecutionFactory class. The use of these subclasses should be considered deprecated in favor of directly using the ExecutionFactory class.

Strategy Classes and Semantic Variation Points

There are two semantic variation points defined for fUML (see 2.3): event dispatch scheduling and polymorphic operation dispatching. In both of these cases, the execution model limits the semantic variability to the behavior of a single operation: ObjectActivation::getNextEvent (see 8.8.1) in the case of event dispatching and Object::dispatch in the case of operation dispatching (see 8.7.1). The execution model uses the Strategy pattern in order to allow for possible variation in the behavior of these operations.

The Strategy pattern involves defining an abstract base strategy class for an operation whose behavior is to be allowed to vary. This base class defines an abstract operation corresponding to the original operation, to which the original operation is delegated. Different concrete subclasses of the base strategy class can then define different concrete behaviors for the operation, and selecting a specific behavior (or strategy) corresponds to using an instance of a specific concrete strategy class.

In the execution model, all strategy classes ultimately descend, directly or indirectly, from the class SemanticStrategy (see Figure 8.4). The SemanticStrategy class provides a common operation for getting the "name" of a strategy, which identifies to which semantic variation point a strategy instance applies. The standard strategy names used in the execution model correspond to the names of the operations whose behavior is being provided: "getNextEvent" and "dispatch."

The strategy to be used for a semantic variation point is determined by the strategy instance that is registered with the execution factory (using the setStrategy operation) at a given locus under the corresponding strategy name. There must be exactly one strategy instance, of the appropriate subclass, registered for each semantic variation point. The execution factory getStrategy operation provides a lookup mechanism for retrieving a strategy instance to be used for a specific named semantic variation.

For further discussion of the strategy classes related to each semantic variation point, as well as the default strategies provided in the execution model, see 8.7.1 and 8.8.1.

Note: While there are currently only two semantic variation points defined for fUML, the strategy mechanism has intentionally been made general enough to accommodate the possible need for additional variation points in future extensions to the specification of the execution semantics for larger subsets of UML.

Specifying Nondeterministic Behavior

There are a number of cases in which the UML 2 Specification specifically indicates that the execution semantics in a certain area are nondeterministic-that is, the semantic specification does not prescribe which one of a number of possible choices is taken during an actual execution. A legal execution may take any one of the allowed choices. For example, if more than one clause of a conditional node has a successful test, then only one of the clause bodies will be executed, but it is nondeterministic which one is actually executed.

In order to model nondeterministic behavior in the execution model, a special case of the Strategy pattern is used. A choice strategy is one with the name "choice" that provides a single operation called choose. This operation takes a single integer argument size (which must be greater than zero) and returns an integer value from 1 to the given size.

The ChoiceStrategy class (see Figure 8.4) is the abstract base strategy class for all choice strategies. A single instance of a concrete subclass of ChoiceStrategy is registered with the execution factory at each locus. Whenever a behavior specification within the execution model is required to make a non-deterministic choice between some number of options, this choice is made by getting the registered choice strategy and using its choose operation.

The key point is that a legal execution may use any choice strategy at all, so long as the "choose operation" always returns a selection from 1 to the required number of choices. Since any choice strategy is legal, no restriction is placed on a conforming execution tool as to how such choices are actually made in its specific implementation. In this way, the concept of nondeterminism operationally is interpreted in the execution model.

For completeness, the execution model includes a single concrete default choice strategy class, FirstChoiceStrategy (see Figure 8.4). The choose operation of this class always returns 1, which corresponds to always picking the first of a list of possible options. It is important to understand that, while this specific strategy is deterministic, the effective nondeterminism of allowed behavior comes about because any other choice strategy might also be used, whether it is some other simple algorithm, totally random or just based on what is most convenient for the internal implementation of some execution tool.

Note: There is no requirement that a conforming execution tool provide a formal specification of what its effective choice strategy is, as this may be entirely implicit in the way the tool is implemented. On the other hand, a specific choice strategy may be formally specified by defining a new subclass of ChoiceStrategy. This may be useful, for example, if the implementation target is in a domain (such as life critical systems) in which fully determinable behavior is desirable or if it is desirable to be able to specify some sort of fair or parameterized distribution of how choices are made.

Primitive Behaviors and Primitive Types

The execution factory at each locus maintains the set of primitive behaviors available to be called by executions at a specific locus. In fUML, primitive behaviors are defined syntactically as instances of OpaqueBehavior. For each OpaqueBehavior instance representing a primitive behavior, the execution factory maintains a corresponding prototype instance of OpaqueBehaviorExecution. When an instance of OpaqueBehavior is passed to the execution factory createExecution operation, the corresponding prototype opaque behavior execution is looked up. A copy of this prototype execution instance is then returned as the result of the createExecution call.

Subclause 9.3 specifies the basic library of primitive behaviors that must be provided by any conforming execution tool. However, specific execution tools may also provide additional primitive behaviors. These are modeled as additional opaque behavior execution prototypes added to the standard list required to be maintained by any execution factory.

Finally, the execution factory also maintains a list of built-in primitive types for which there are corresponding literal value specifications. Note that this is a list of instances of the PrimitiveType metaclass – that is, representations of the M1-level types from the fUML model library (see 9.2). During the evaluation of a literal value specification, the appropriate evaluation class looks up by name the proper primitive type to attach to the resulting value (see 8.4). Since fUML includes literal value specifications for Boolean, Integer, Real, String, and UnlimitedNatural (see 7.4), the list of built-in types must include at least these types.

Configuring the Execution Environment at a Locus

While the Executor class provides the basic interface for evaluating value specifications and executing behaviors, the preceding discussion in this subclause indicates that more than just an instance of an executor is required in order to even begin to perform such evaluations and executions. Instead, it is necessary to instantiate a set of collaborating objects (largely from classes within the execution model) that provide the initial execution environment. The configuration of this initial environment in terms of the execution model is an abstraction of the capabilities that a conforming execution tool must actually provide in order to execute a fUML model.

The following items are required as part of the execution environment at a specific locus.

- A single instance of class Locus. The identifier for this instance must be initialized to a non-empty string, which should be distinct from the identifiers for any remove loci that may be available from this locus.
- A single instance of class Executor, linked to the locus.
- A single instance of ExecutionFactory, also linked to the locus (see 8.3.2.2).
- Instances of PrimitiveType for each of the primitive types Boolean, Integer, Real, String and UnlimitedNatural, as defined in the Foundational Model Library (see 9.2), registered with the execution factory as built-in types.
- Single instances of concrete subclasses of ChoiceStrategy (see 8.3.2.1), DispatchStrategy (see 8.7.2.1) and GetNextEventStrategy (see 8.8.2.9), all registered with the execution factory.

The following items are also permitted as part of the execution environment at a specific locus.

- Instances of concrete subclasses of OpaqueBehaviorExecution registered with the execution factory as primitive behavior prototypes (these may include some or all of the primitive behaviors from the Foundational Model Library (see 9.3).
- Instances of Object representing discoverable services, instantiated as existential values at the locus (these may include singleton instances of the basic input/output classes StandardInputChannel and StandardOutputChannel (see 9.5).

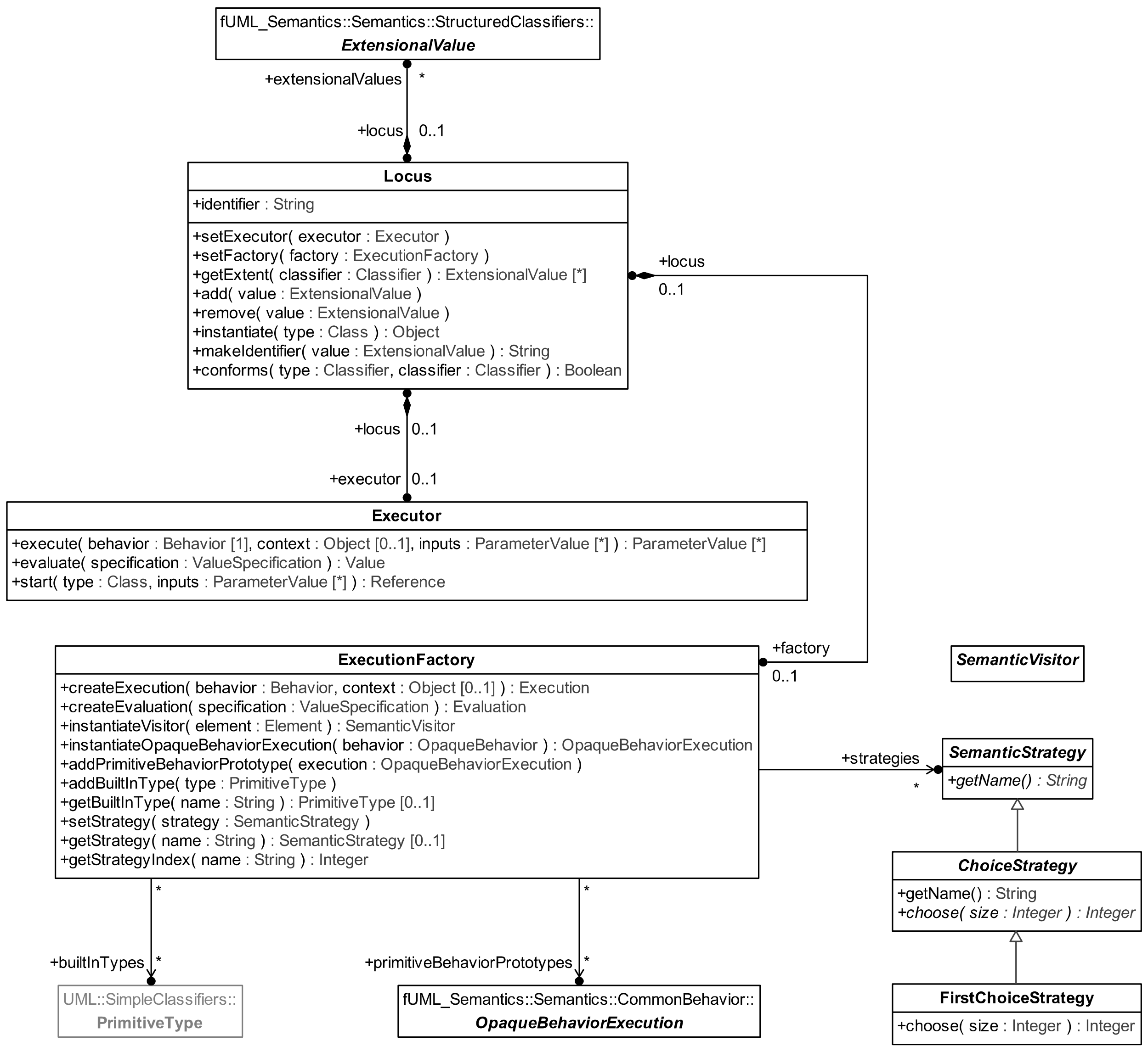

fUML_Semantics::Semantics::StructuredClassifiers::
ExtensionalValue
+extensionalValues *
+locus 0..1
Locus
+identifier : String
+setExecutor(executor : Executor)
+setFactory(factory : ExecutionFactory)
+getExtent(classifier : Classifier) : ExtensionalValue [*]
+add(value : ExtensionalValue)
+remove(value : ExtensionalValue)
+instantiate(type : Class) : Object
+makeIdentifier(value : ExtensionalValue) : String
+conforms(type : Classifier, classifier : Classifier) : Boolean
+locus 0..1
+locus 0..1
+executor 0..1
Executor
+execute(behavior : Behavior [1], context : Object [0..1], inputs : ParameterValue [*]) : ParameterValue [*]
+evaluate(specification : ValueSpecification) : Value
+start(type : Class, inputs : ParameterValue [*]) : Reference
ExecutionFactory
+createExecution(behavior : Behavior, context : Object [0..1]) : Execution
+createEvaluation(specification : ValueSpecification) : Evaluation
+instantiateVisitor(element : Element) : SemanticVisitor
+instantiateOpaqueBehaviorExecution(behavior : OpaqueBehavior) : OpaqueBehaviorExecution
+addPrimitiveBehaviorPrototype(execution : OpaqueBehaviorExecution)
+addBuiltInType(type : PrimitiveType)
+getBuiltInType(name : String) : PrimitiveType [0..1]
+setStrategy(strategy : SemanticStrategy)
+getStrategy(name : String) : SemanticStrategy [0..1]
+getStrategyIndex(name : String) : Integer
+factory 0..1
SemanticVisitor
+strategies *
SemanticStrategy
+getName() : String
ChoiceStrategy
+getName() : String
+choose(size : Integer) : Integer
FirstChoiceStrategy
+choose(size : Integer) : Integer
* +builtInTypes *
* +primitiveBehaviorPrototypes *
UML::SimpleClassifiers::
PrimitiveType
fUML_Semantics::Semantics::CommonBehavior::
OpaqueBehaviorExecution

Figure 8.4 - Loci

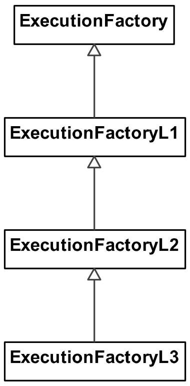


Figure 8.5 - Execution Factories (use of subclasses of ExecutionFactory is deprecated)

8.3.2 Class Descriptions

8.3.2.1 ChoiceStrategy

A choice strategy is used to represent the behavior of making an arbitrary non-deterministic choice.
A valid execution may use ANY choice strategy for choosing one element from a given set.

Generalizations

- SemanticStrategy

Attributes

None

Associations

None

Operations

[1] choose (in size : Integer) : Integer

```
Choose an integer from 1 to the given size.
[The size must be greater than 0.]
```

[2] getName ( ) : String

```
// The name of a choice strategy is always "choice".
return "choice";
```

8.3.2.2 ExecutionFactory

An execution factory is used to create objects that represent the execution of a behavior, the evaluation of a value specification or the activation of an activity node.

Generalizations

None

Attributes

None

Associations

- builtInTypes : PrimitiveType [0..*]
  The set of primitive types that have corresponding literal value specifications.
  Must include Integer, Boolean, String, and UnlimitedNatural.
- locus : Locus [0..1]
  The locus at which this factory resides.
- primitiveBehaviorPrototypes : OpaqueBehaviorExecution [0..*]
  The set of opaque behavior executions to be used to execute the primitive behaviors known to the factory.
- strategies : SemanticStrategy [0..*]
  The set of semantic strategies currently registered with this execution factory.

Operations

[1] addBuiltInType (in type : PrimitiveType)

```
// Add the given primitive type as a built-in type.
// Precondition: No built-in type with the same name should already exist.

this.builtInTypes.addValue(type);
```

[2] addPrimitiveBehaviorPrototype (in execution : OpaqueBehaviorExecution)

```
// Add an opaque behavior execution to use as a prototype for instantiating the corresponding
primitive opaque behavior.
// Precondition: No primitive behavior prototype for the type of the given execution should
already exist.

this.primitiveBehaviorPrototypes.addValue(execution);
```

[3] createEvaluation (in specification : ValueSpecification) : Evaluation

```
// Create an evaluation object for a given value specification.
// The evaluation will take place at the locus of the factory.

Evaluation evaluation = (Evaluation)(this.instantiateVisitor(specification));
```

```
evaluation.specification = specification;
evaluation.locus = this.locus;

return evaluation;
```

[4] createExecution (in behavior : Behavior, in context : Object [0..1]) : Execution

```
// Create an execution object for a given behavior.
// The execution will take place at the locus of the factory in the given context.
// If the context is empty, the execution is assumed to provide its own context.

Execution execution;

if (behavior instanceof OpaqueBehavior) {
    execution = this.instantiateOpaqueBehaviorExecution((OpaqueBehavior)behavior);
}
else {
    execution = (Execution)(this.instantiateVisitor(behavior));
    execution.types.addValue(behavior);
    execution.createFeatureValues();
}

this.locus.add(execution);

if (context == null) {
    execution.context = execution;
}
else {
    execution.context = context;
}

return execution;
```

[5] getBuiltInType (in name : String) : PrimitiveType [0..1]

```
// Return the built-in type with the given name.

PrimitiveType type = null;
int i = 1;
while (type == null & i <= this.builtInTypes.size()) {
    PrimitiveType primitiveType = this.builtInTypes.getValue(i-1);
    if (primitiveType.name.equals(name)) {
        type = primitiveType;
    }
    i = i + 1;
}

return type;
```

[6] getStrategy (in name : String) : SemanticStrategy [0..1]

```
// Get the strategy with the given name.

int i = this.getStrategyIndex(name);

SemanticStrategy strategy = null;
if (i <= this.strategies.size()) {
     strategy = this.strategies.getValue(i-1);
}

return strategy;
```

[7] getStrategyIndex (in name : String) : Integer

```
// Get the index of the strategy with the given name.
// If there is no such strategy, return the size of the strategies list.

SemanticStrategyList strategies = this.strategies;

int i = 1;
boolean unmatched = true;
while (unmatched & (i <= strategies.size())) {
     if (strategies.getValue(i-1).getName().equals(name)) {
         unmatched = false;
    } else {
         i = i + 1;
    }
}

return i;
```

[8] instantiateOpaqueBehaviorExecution (in behavior : OpaqueBehavior) : OpaqueBehaviorExecution

```
// Return a copy of the prototype for the primitive behavior execution of the given opaque
behavior.

OpaqueBehaviorExecution execution = null;
int i = 1;
while (execution == null & i <= this.primitiveBehaviorPrototypes.size()) {
     OpaqueBehaviorExecution prototype = this.primitiveBehaviorPrototypes.getValue(i-1);
     if (prototype.getBehavior() == behavior) {
          execution = (OpaqueBehaviorExecution)(prototype.copy());
   }
   i = i + 1;
}

if (execution == null) {
}

return execution;
```

[9] instantiateVisitor (in element : Element, in suffix : String) : SemanticVisitor

```
// Instantiate a visitor object for the given element.

SemanticVisitor visitor = null;

// Formerly Level L1

if (element instanceof LiteralBoolean) {
     visitor = new LiteralBooleanEvaluation();
}

else if (element instanceof LiteralString) {
     visitor = new LiteralStringEvaluation();
}

else if (element instanceof LiteralNull) {
     visitor = new LiteralNullEvaluation();
}

else if (element instanceof InstanceValue) {
     visitor = new InstanceValueEvaluation();
}

else if (element instanceof LiteralUnlimitedNatural) {
     visitor = new LiteralUnlimitedNaturalEvaluation();
}

else if (element instanceof LiteralInteger) {
     visitor = new LiteralIntegerEvaluation();
}
```

```
else if (element instanceof LiteralReal) {
     visitor = new LiteralRealEvaluation();
}

else if (element instanceof CallEventBehavior) {
     visitor = new CallEventExecution();

// Formerly Level L2

} else if (element instanceof Activity) {
     visitor = new ActivityExecution();
}

else if (element instanceof ActivityParameterNode) {
     visitor = new ActivityParameterNodeActivation();
}

else if (element instanceof CentralBufferNode &
          !{element instanceof DataStoreNode)) {
     visitor = new CentralBufferNodeActivation();
}

else if (element instanceof InitialNode) {
     visitor = new InitialNodeActivation();
}

else if (element instanceof ActivityFinalNode) {
     visitor = new ActivityFinalNodeActivation();
}

else if (element instanceof FlowFinalNode) {
     visitor = new FlowFinalNodeActivation();
}

else if (element instanceof JoinNode) {
     visitor = new JoinNodeActivation();
}

else if (element instanceof MergeNode) {
     visitor = new MergeNodeActivation();
}

else if (element instanceof ForkNode) {
     visitor = new ForkNodeActivation();
}

else if (element instanceof DecisionNode) {
     visitor = new DecisionNodeActivation();
}

else if (element instanceof InputPin) {
     visitor = new InputPinActivation();
}

else if (element instanceof OutputPin) {
     visitor = new OutputPinActivation();
}

else if (element instanceof CallBehaviorAction) {
     visitor = new CallBehaviorActionActivation();
}

else if (element instanceof CallOperationAction) {
     visitor = new CallOperationActionActivation();
}

else if (element instanceof SendSignalAction) {
     visitor = new SendSignalActionActivation();
}

else if (element instanceof ReadSelfAction) {
     visitor = new ReadSelfActionActivation();
}

else if (element instanceof TestIdentityAction) {
     visitor = new TestIdentityActionActivation();
}

else if (element instanceof ValueSpecificationAction) {
     visitor = new ValueSpecificationActionActivation();
}
```

```
else if (element instanceof CreateObjectAction) {
     visitor = new CreateObjectActionActivation();
}

else if (element instanceof DestroyObjectAction) {
     visitor = new DestroyObjectActionActivation();
}

else if (element instanceof ReadStructuralFeatureAction) {
     visitor = new ReadStructuralFeatureActionActivation();
}

else if (element instanceof ClearStructuralFeatureAction) {
     visitor = new ClearStructuralFeatureActionActivation();
}

else if (element instanceof AddStructuralFeatureValueAction) {
     visitor = new AddStructuralFeatureValueActionActivation();
}

else if (element instanceof RemoveStructuralFeatureValueAction) {
     visitor = new RemoveStructuralFeatureValueActionActivation();
}

else if (element instanceof ReadLinkAction) {
     visitor = new ReadLinkActionActivation();
}

else if (element instanceof ClearAssociationAction) {
     visitor = new ClearAssociationActionActivation();
}

else if (element instanceof CreateLinkAction) {
     visitor = new CreateLinkActionActivation();
}

else if (element instanceof DestroyLinkAction) {
     visitor = new DestroyLinkActionActivation();
}
```

```
// Formerly Level L3

else if (element instanceof DataStoreNode) {
     visitor = new DataStoreNodeActivation();
}

else if (element instanceof ConditionalNode) {
     visitor = new ConditionalNodeActivation();
}

else if (element instanceof LoopNode) {
     visitor = new LoopNodeActivation();
}

else if (element instanceof ExpansionRegion) {
     visitor = new ExpansionRegionActivation();
}

// Note: Since ConditionalNode, LoopNode and ExpansionRegion are
// subclasses of StructuredActivityNode, element must be tested
// against the three subclasses before the superclass.
else if (element instanceof StructuredActivityNode) {
     visitor = new StructuredActivityNodeActivation();
}

else if (element instanceof ExpansionNode) {
     visitor = new ExpansionNodeActivation();
}

else if (element instanceof ReadExtentAction) {
     visitor = new ReadExtentActionActivation();
}

else if (element instanceof ReadIsClassifiedObjectAction) {
     visitor = new ReadIsClassifiedObjectActionActivation();
}

else if (element instanceof ReclassifyObjectAction) {
     visitor = new ReclassifyObjectActionActivation();
}
```

```
else if (element instanceof StartObjectBehaviorAction) {
     visitor = new StartObjectBehaviorActionActivation();
}

else if (element instanceof StartClassifierBehaviorAction) {
     visitor = new StartClassifierBehaviorActionActivation();
}

// Note: Since AcceptCallAction is a subclass of AcceptEventAction,
// element must be tested against AcceptCallAction before
//  AcceptEventAction.
else if (element instanceof AcceptCallAction) {
     visitor = new AcceptCallActionActivation();
}

else if (element instanceof AcceptEventAction) {
     visitor = new AcceptEventActionActivation();
}

else if (element instanceof ReplyAction) {
     visitor = new ReplyActionActivation();
}

else if (element instanceof ReduceAction) {
     visitor = new ReduceActionActivation();
}
else if (element instanceof RaiseExceptionAction) {
    visitor = new RaiseExceptionActionActivation();
}

return visitor;
```

[10] setStrategy (in strategy : SemanticStrategy)

```
// Set the strategy for a semantic variation point. Any existing strategy for the same SVP is
replaced.

int i = this.getStrategyIndex(strategy.getName());

if (i <= this.strategies.size()) {
     this.strategies.removeValue(i-1);
}
```

```
this.strategies.addValue(strategy);
```

8.3.2.3 ExecutionFactoryL1 (Deprecated)

This subclass is provided for backward compatibility with previous versions of fUML. Its use is deprecated.

8.3.2.4 ExecutionFactoryL2 (Deprecated)

This subclass is provided for backward compatibility with previous versions of fUML. Its use is deprecated.

8.3.2.5 ExecutionFactoryL3 (Deprecated)

This subclass is provided for backward compatibility with previous versions of fUML. Its use is deprecated.

8.3.2.6 Executor

An executor is used to execute behaviors and evaluation value specifications.

Generalizations

None

Attributes

None

Associations

- locus : Locus [0..1]
  The locus at which this executor resides.

Operations

[1] evaluate (in specification : ValueSpecification) : Value

```
// Evaluate the given value specification, returning the specified value.

return  this.locus.factory.createEvaluation(specification).evaluate();
```

[2] execute (in behavior : Behavior, in context : Object [0..1], in inputs : ParameterValue [0..*]) : ParameterValue [0..*]

```
// Execute the given behavior with the given input values in the given context, producing the given output values.
// There must be one input parameter value for each input (in or in-out) parameter of the behavior.
// The returned values include one parameter value for each output (in-out, out or return) parameter of the behavior.
// The execution instance is destroyed at completion.

Execution execution = this.locus.factory.createExecution(behavior, context);
```

```
for (int i = 0; i < inputs.size(); i++) {
     execution.setParameterValue(inputs.getValue(i));
}

execution.execute();
ParameterValueList outputValues = execution.getOutputParameterValues();
execution.destroy();

return outputValues;
```

[3] start (in type : Class, in inputs : ParameterValue [0..*]) : Reference

```
// Instantiate the given class and start any behavior of the resulting object.
// (The behavior of an object includes any classifier behaviors for an active object or the
class of the object itself, if that is a behavior.)

Object_ object = this.locus.instantiate(type);

object.startBehavior(type, inputs);

Reference reference = new Reference();
reference.referent = object;

return reference;
```

8.3.2.7 FirstChoiceStrategy

Generalizations

- ChoiceStrategy

Attributes

None

Associations

None

Operations

[1] choose (in size : Integer) : Integer

```
// Always choose one.

return 1;
```

8.3.2.8 Locus

A locus is a place at which extensional values (objects or links) can exist. The extent of a class or association is the set of objects or links of that type that exist at a certain locus.

A locus also has an executor and a factory associated with it, used to execute behaviors as a result of requests dispatched to objects at the locus.

Generalizations

None

Attributes

- identifier : String
  The identifier of this locus, which should be unique at least within the current execution environment.

Associations

- executor : Executor [0..1]
  The executor to be used at this locus.
- extensionalValues : ExtensionalValue [0..*]
  The set of values that are members of classifier extents at this locus.
- factory : ExecutionFactory [0..1]
  The factory to be used at this locus.

Operations

[1] add (in value : ExtensionalValue)

```
// Add the given extensional value to this locus

value.locus = this;
value.identifier = this.identifier + "#" + this.makeIdentifier(value);
this.extensionalValues.addValue(value);
```

[2] conforms (in type : Classifier, in classifier : Classifier) : Boolean

```
// Test if a type conforms to a given classifier, that is, the type is equal to or a
descendant of the classifier.

boolean doesConform = false;

if (type == classifier) {
     doesConform = true;
} else {
     int i = 1;
     while (!doesConform & i <= type.general.size()) {
          doesConform = this.conforms(type.general.getValue(i-1), classifier);
          i = i + 1;
```

```
    }
}

return doesConform;
```

[3] getExtent (in classifier : Classifier) : ExtensionalValue [0..*]

```
// Return the set of extensional values at this locus which have the given classifier as a
type.

ExtensionalValueList extent = new ExtensionalValueList();

ExtensionalValueList extensionalValues = this.extensionalValues;
for (int i = 0; i < extensionalValues.size(); i++) {
    ExtensionalValue value = extensionalValues.getValue(i);
    ClassifierList types = value.getTypes();

    boolean conforms = false;
    int j = 1;
    while (!conforms & j <= types.size()) {
        conforms = this.conforms(types.getValue(j-1), classifier);
        j = j + 1;
    }

    if (conforms) {
        extent.addValue(value);
    }
}

return extent;
```

[4] instantiate (in type : Class) : Object

```
// Instantiate the given class at this locus.

Object_ object = null;

if (type instanceof Behavior) {
    object = this.factory.createExecution((Behavior)type, null);
}
else {
    object = new Object_();
```

```
     object.types.addValue(type);
     object.createFeatureValues();
     this.add(object);
}

return object;
```

[5] makeIdentifier (in value : ExtensionalValue) : String

```
// Return an identifier for the given (newly created) extensional value.

// [No normative specification. A conforming implementation may create an identifier
// in any way such that all identifiers for extensional values created at any one
// locus are unique.]
```

[5] remove (in value : ExtensionalValue)

```
// Remove the given extensional value from this locus.

value.locus = null;

boolean notFound = true;
int i = 1;
while (notFound & i <= this.extensionalValues.size()) {
     if (this.extensionalValues.getValue(i-1) == value) {
          this.extensionalValues.remove(i-1);
          notFound = false;
     }
     i = i + 1;
}
```

[6] setExecutor (in executor : Executor)

```
// Set the executor for this locus.

this.executor = executor;
this.executor.locus = this;
```

[7] setFactory (in factory : ExecutionFactory)

```
// Set the factory for this locus.

this.factory = factory;
this.factory.locus = this;
```

8.3.2.9 SemanticStrategy

The common base class for semantic strategy classes. A semantic strategy class specifies the behavior to be used at a specific semantic variation point.

Generalizations

None

Attributes

None

Associations

None

Operations

[1] getName ( ) : String

```
Return the name of this strategy, as defined for the semantic variation point to which the
strategy applies.
```

8.3.2.10 SemanticVisitor

The common base class for semantic visitor classes.

Generalizations

None

Attributes

None

Associations

None

Operations

[1] _beginIsolation ( )

[2] _endIsolation ( )

8.4 Values

8.4.1 Overview

Values

As discussed in 6.2, a model is interpreted to make statements about some semantic domain. First order statements are actually made on instances in the semantics domain. The structural semantics of UML provides the denotational mapping of appropriate UML model elements to such semantic instances.

The term instance is often used to mean an object of a specific class. However, in UML, this needs to be generalized to the concept of an instance of any classifier. The appropriate UML model elements for representing this generalized concept are value specifications.

Figure 7.7 in 7.4 and Figure 7.12 in 7.5 show the subset of the abstract syntax of UML value specifications that is included in fUML. This subset includes the syntax for model elements representing literals of primitive types such as integers and Booleans, as well as instances of structured types, which include non-primitive data types and classes.

The denotation of a value specification is given formally by the evaluate operation of the Executor class (see 8.3). This operation maps an instance of the abstract syntax type ValueSpecification to an instance of the semantic type Value. Just as the abstract syntax of UML can itself be modeled in UML, the semantic domain for UML can also be modeled in UML. Figure 8.3 shows this model for Value.

Clearly, literal specifications map to primitive values: literal integers to integer values, literal Booleans to Boolean values, etc. The mapping for instance values is not so straightforward. An instance value is the specification of a value as an instance of a non-primitive classifier. The classifier may be an enumeration, a structured data type or a class. Such value specifications map to enumeration and structured values. The instances of simple classifiers (primitive types, enumerations and data types) are discussed further in 8.6, while instance of classes and associations are described in 8.7.

Consider, for example, the simple instance model from Figure 6.1 in 6.2. Figure 8.6 gives the representation of this model in terms of the abstract syntax of ValueSpecification. The result of the operation evaluate acting on the instance value v (a kind of ValueSpecification) in Figure 8.7 is then the object j (a kind of structured value) given in Figure 8.6.

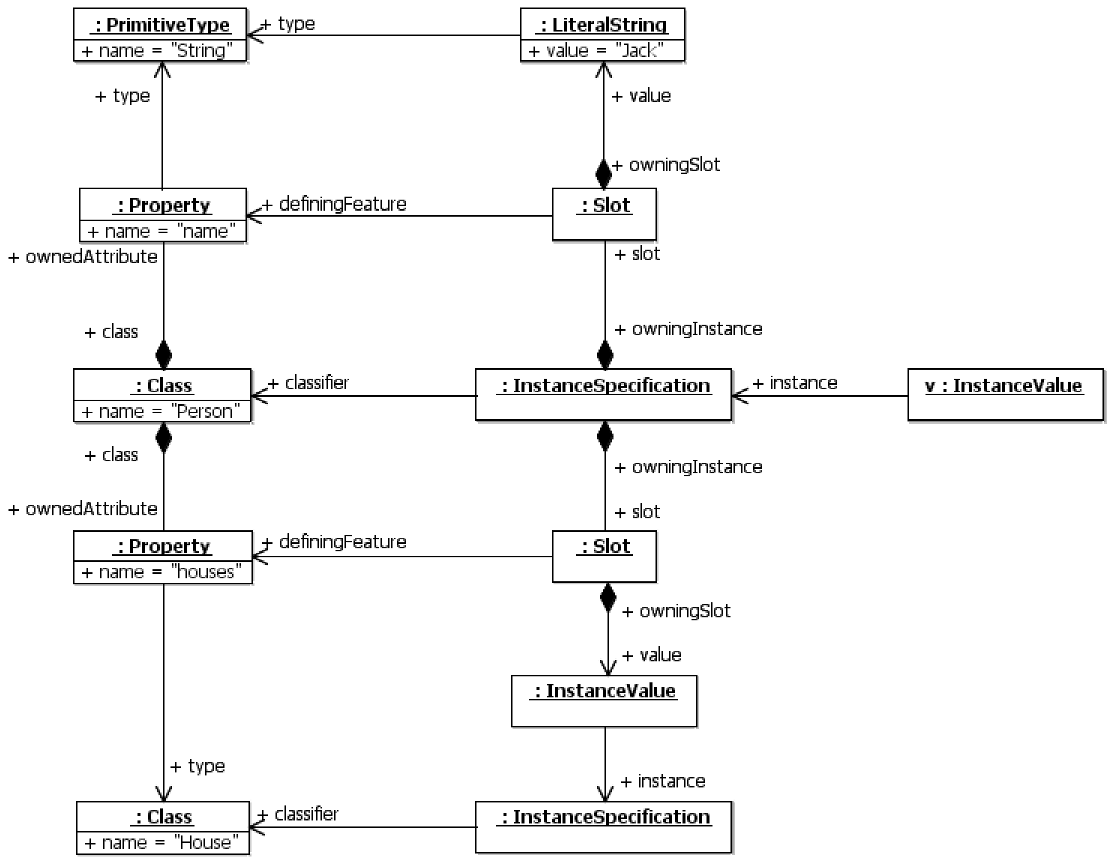


Figure 8.6 - Abstract Syntax Representation of a Simple Instance Model

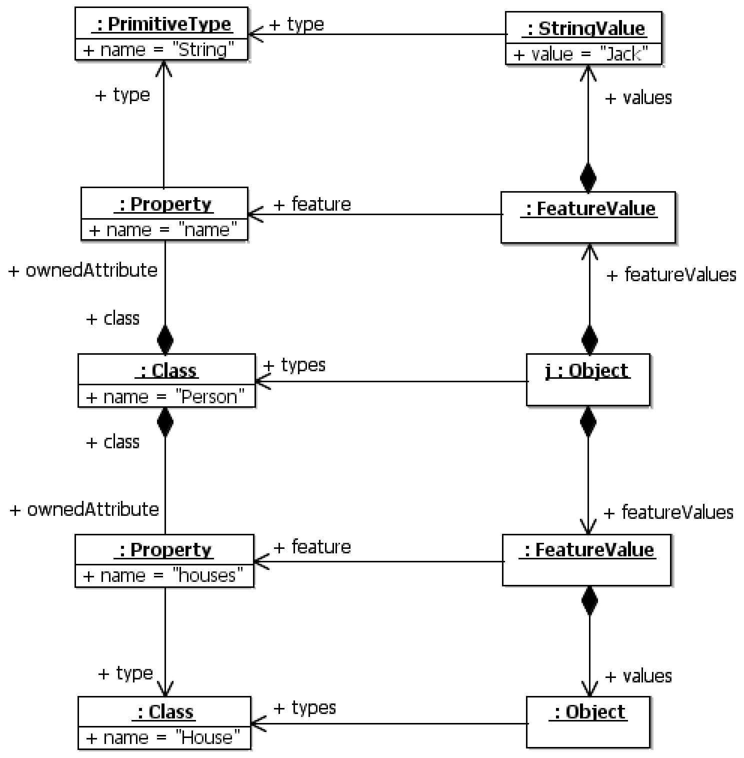


Figure 8.7 - Semantic Interpretation of a Simple Instance Model

Evaluations

An evaluation is a kind of visitor class used to evaluate value specifications (see 8.3 for a general discussion of visitor classes). As shown in Figure 8.9, there is an evaluation class corresponding to each concrete subclass of the abstract syntax metaclass ValueSpecification (instance values are covered in 8.5).

To evaluate a value specification, the executor uses the execution factory to create an instance of the appropriate evaluation class (see 8.3), with a reference to the representation of the value specification to be evaluated. Evaluation is actually carried out by calling the evaluate method on the evaluation object, which then returns a value of the appropriate type.

An evaluation object is also created with a reference to the execution locus. This provides access to the execution factory at the locus in order to obtain the proper primitive type to use for the value resulting from a literal evaluation.

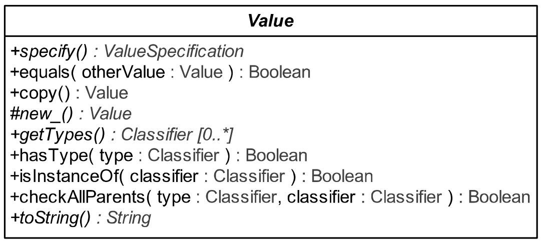

Value
+specify() : ValueSpecification
+equals(otherValue : Value) : Boolean
+copy() : Value
#new_() : Value
+getTypes() : Classifier [0..*]
+hasType(type : Classifier) : Boolean
+isInstanceOf(classifier : Classifier) : Boolean
+checkAllParents(type : Classifier, classifier : Classifier) : Boolean
+toString() : String

Figure 8.8 - Values

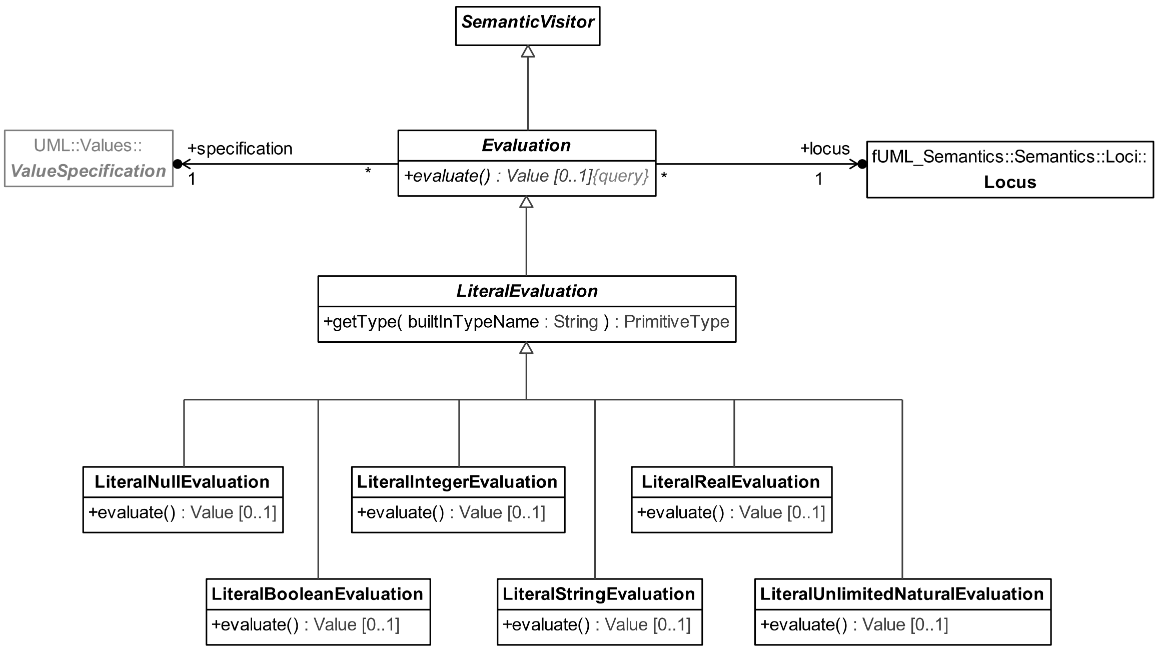

SemanticVisitor
UML::Values::
ValueSpecification
+specification
1
*
Evaluation
+evaluate() : Value [0..1]{query}
*
+locus
1
fUML_Semantics::Semantics::Loci::
Locus
LiteralEvaluation
+getType(builtInTypeName : String) : PrimitiveType
LiteralNullEvaluation
+evaluate() : Value [0..1]
LiteralIntegerEvaluation
+evaluate() : Value [0..1]
LiteralRealEvaluation
+evaluate() : Value [0..1]
LiteralBooleanEvaluation
+evaluate() : Value [0..1]
LiteralStringEvaluation
+evaluate() : Value [0..1]
LiteralUnlimitedNaturalEvaluation
+evaluate() : Value [0..1]

Figure 8.9 - Evaluations

8.4.2 Class Descriptions

8.4.2.1 Evaluation

An evaluation is used to evaluate a value specification to produce a value.

Generalizations

- SemanticVisitor

Attributes

None

Associations

- locus : Locus
  The locus at which this evaluation is taking place.
- specification : ValueSpecification
  The value specification to be evaluated.

Operations

[1] evaluate ( ) : Value [0..1]

```
Evaluate the specification, returning the resulting value.
```

8.4.2.2 LiteralBooleanEvaluation

A boolean evaluation is an evaluation whose specification is a literal boolean.

Generalizations

- LiteralEvaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// Evaluate a literal boolean, producing a boolean value.

LiteralBoolean literal = (LiteralBoolean)specification;
BooleanValue booleanValue = new BooleanValue();
booleanValue.type = this.getType("Boolean");
booleanValue.value = literal.value;

return booleanValue;
```

8.4.2.3 LiteralEvaluation

A literal evaluation is an evaluation whose specification is a Literal Specification.

Generalizations

- Evaluation

Attributes

None

Associations

None

Operations

[1] getType (in builtInTypeName : String) : PrimitiveType

```
// Get the type of the specification. If that is null, then use the built-in type of the
given name.

PrimitiveType type = (PrimitiveType)(this.specification.type);

if (type == null) {
     type = this.locus.factory.getBuiltInType(builtInTypeName);
}

return type;
```

8.4.2.4 LiteralIntegerEvaluation

A literal integer evaluation is an evaluation whose specification is a literal integer.

Generalizations

- LiteralEvaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// Evaluate a literal integer, producing an integer value.

LiteralInteger literal = (LiteralInteger)specification;
IntegerValue integerValue = new IntegerValue();
integerValue.type = this.getType("Integer");
integerValue.value = literal.value;

return integerValue;
```

8.4.2.5 LiteralNullEvaluation

A literal null evaluation is an evaluation whose specification is a literal null.

Generalizations

- LiteralEvaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// Evaluate a literal null, returning nothing (since a null represents an "absence of any
value").

return null;
```

8.4.2.6 LiteralRealEvaluation

A literal real evaluation is an evaluation whose specification is a literal real.

Generalizations

- LiteralEvaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// Evaluate a literal real, producing a real value.

LiteralReal literal = (LiteralReal)specification;
RealValue realValue = new RealValue();
realValue.type = this.getType("Real");
realValue.value = literal.value;
return realValue;
```

8.4.2.7 LiteralStringEvaluation

A literal string evaluation is an evaluation whose specification is a literal string.

Generalizations

- LiteralEvaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// Evaluate a literal string, producing a string value.

LiteralString literal = (LiteralString)specification;
StringValue stringValue = new StringValue();
stringValue.type = this.getType("String");
stringValue.value = literal.value;

return stringValue;
```

8.4.2.8 LiteralUnlimitedNaturalEvaluation

A literal unlimited natural evaluation is an evaluation whose specification is a literal unlimited natural.

Generalizations

- LiteralEvaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// Evaluate a literal unlimited natural producing an unlimited natural value.

LiteralUnlimitedNatural literal = (LiteralUnlimitedNatural)specification;
```

```
UnlimitedNaturalValue unlimitedNaturalValue = new UnlimitedNaturalValue();
unlimitedNaturalValue.type = this.getType("UnlimitedNatural");
unlimitedNaturalValue.value = literal.value;

return unlimitedNaturalValue;
```

8.4.2.9 Value

A value is an instance of one or more classifiers, which are its types. A value is always representable using a value specification.

[**Note:** Value specializes SemanticVisitor to allow the Execution subclass to be a semantic visitor, without requiring multiple generalization of Execution.]

Generalizations

- SemanticVisitor

Attributes

None

Associations

None

Operations

[1] checkAllParents (in type : Classifier, in classifier : Classifier) : Boolean

```
// Check if the given classifier matches any of the direct or indirect
// ancestors of a given type.

ClassifierList directParents = type.general;
boolean matched = false;
int i = 1;
while (!matched & i <= directParents.size()) {
     Classifier directParent = directParents.getValue(i - 1);
     if (directParent == classifier) {
          matched = true;
     } else {
          matched = this.checkAllParents(directParent, classifier);
     }
     i = i + 1;
}

return matched;
```

[2] copy ( ) : Value

```
// Create a new value that is equal to this value.
// By default, this operation simply creates a new value with empty properties.
// It must be overridden in each Value subclass to do the superclass copy and then
appropriately set properties defined in the subclass.

return this.new_();
```

[3] equals (in otherValue : Value) : Boolean

```
// Test if this value is equal to otherValue. To be equal, this value must have the same type
as otherValue.
// This operation must be overridden in Value subclasses to check for equality of properties
defined in those subclasses.

ClassifierList myTypes = this.getTypes();
ClassifierList otherTypes = otherValue.getTypes();

boolean isEqual = true;

if (myTypes.size() != otherTypes.size()) {
    isEqual = false;

} else {
    int i = 1;
    while (isEqual & i <= myTypes.size()) {

        boolean matched = false;
        int j = 1;
        while (!matched & j <= otherTypes.size()) {
            matched = (otherTypes.getValue(j-1) == myTypes.getValue(i-1));
            j = j + 1;
        }

        isEqual = matched;
        i = i + 1;
    }
}

return isEqual;
```

[4] getTypes ( ) : Classifier [0..*]

```
Gets all the classifiers under which this value is currently classifier.
```

[5] hasType (in type : Classifier) : Boolean

```
// Check if this object has the given classifier as a type.

ClassifierList types = this.getTypes();

boolean found = false;
int i = 1;
while (!found & i <= types.size()) {
     found = (types.getValue(i-1) == type);
     i = i + 1;
}

return found;
```

[6] isInstanceOf (in classifier : Classifier) : Boolean

```
// Check if this value has the given classifier as its type
// or as an ancestor of one of its types.

ClassifierList types = this.getTypes();

boolean isInstance = this.hasType(classifier);
int i = 1;
while (!isInstance & i <= types.size()) {
     isInstance = this.checkAllParents(types.getValue(i-1), classifier);
     i = i + 1;
}

return isInstance;
```

[7] new_ ( ) : Value

```
Create a new value of the same Value subclass as this value, with all properties empty (even
if this violates multiplicity constraints).
This operation must be defined in each concrete Value subclass to create an instance of that
subclass.
```

[8] specify ( ) : ValueSpecification

```
Return a value specification whose evaluation gives a value equal to this value.
```

[9] toString ( ) : String

```
Return a string representation of this value.
```

8.5 Classification

8.5.1 Overview

An instance value is a value specification used to specify the instance of a classifier based on an instance specification. In fUML, instance values are used to represent instances of non-primitive data types, including enumerations, structured data types and classes (see 7.5 and 7.6). As shown in Figure 8.10, an instance value evaluation is used to evaluate an instance value, producing enumeration value, a compound value or a reference. Enumeration and structured values are described in 8.6, and references are described in 8.7.

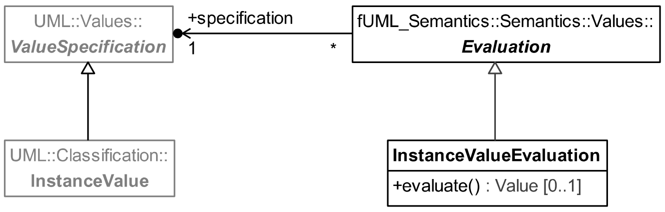


Figure 8.10 - Instance Values

8.5.2 Class Descriptions

8.5.2.1 InstanceValueEvaluation

An instance value evaluation is an evaluation whose specification is an instance value.

Generalizations

- Evaluation

Attributes

None

Associations

None

Operations

[1] evaluate ( ) : Value [0..1]

```
// If the instance specification is for an enumeration, then return the identified
enumeration literal.
// If the instance specification is for a data type (but not a primitive value or an
enumeration), then create a data value of the given data type.
```

```
// If the instance specification is for an object, then create an object at the current locus
with the specified types.
// Set each feature of the created value to the result of evaluating the value specifications
for the specified slot for the feature.

InstanceSpecification instance = ((InstanceValue)this.specification).instance;
ClassifierList types = instance.classifier;
Classifier myType = types.getValue(0);

Value value;
if (instance instanceof EnumerationLiteral) {
     EnumerationValue enumerationValue = new EnumerationValue();
     enumerationValue.type = (Enumeration)myType;
     enumerationValue.literal = (EnumerationLiteral)instance;
     value = enumerationValue;
}
else {

     StructuredValue structuredValue = null;

     if (myType instanceof DataType) {
          DataValue dataValue = new DataValue();
          dataValue.type = (DataType)myType;
          structuredValue = dataValue;
     }
     else {
          Object_ object = null;
          if (myType instanceof Behavior) {
               object = this.locus.factory.createExecution((Behavior)myType, null);
          }
          else {
               object = new Object_();
               for (int i = 0; i < types.size(); i++) {
                    Classifier type = types.getValue(i);
                    object.types.addValue((Class_)type);
               }
          }

          this.locus.add(object);

          Reference reference = new Reference();
```

```
            reference.referent = object;
            structuredValue = reference;
        }

        structuredValue.createFeatureValues();

        SlotList instanceSlots = instance.slot;
        for (int i = 0; i < instanceSlots.size(); i++) {
            Slot slot = instanceSlots.getValue(i);
            ValueList values = new ValueList();

            ValueSpecificationList slotValues = slot.value;
            for (int j = 0; j < slotValues.size(); j++) {
                ValueSpecification slotValue = slotValues.getValue(j);
                values.addValue(this.locus.executor.evaluate(slotValue));
            }
            structuredValue.setFeatureValue(slot.definingFeature, values, 0);
        }

    value = structuredValue;
}

return value;
```

8.6 Simple Classifiers

8.6.1 Overview

Simple Values

The possible values of a primitive type or enumeration are essentially fully determined by the definition of the type. For example, the set of possible values of the primitive type Integer is the mathematical set of integers. While this set is infinite, it is completely specified by its mathematical definition. One cannot “create” a “new” instance of Integer that does not denote an integer value already in the set. In some sense, all the possible instances of Integer are considered to already exist, even though, of course, only a small finite subset of them will be denoted in any given model.

As shown in Figure 8.11, the primitive values are represented as subclasses of Value with an underlying value drawn from the base semantic representation of primitive types (see 10.3.1). The equality of two primitive values of the same type is determined by the equality of their underlying base primitive values, and they have no identity separate from those underlying values.

The possible values for an enumeration, on the other hand, are an explicitly-specified, finite set, denoted by the enumeration literals of the enumeration. As shown in Figure 8.11, an enumeration value is a value associated with a specific enumeration literal of the enumeration. Two enumeration values are equal if they represent the same enumeration literal, so two enumeration values for the same literal are semantically representations of the same value.

Compound values

A structured data type is a data type that is not a primitive type or an enumeration value. An instance value of a structured data type maps to a data value, as shown in Figure 8.12. A data value is a kind of compound value, which associates values with the attributes of the data type. The equality of two data values of the same type is determined by the equality of the values of their attributes. They have no identity separately from their value and are, therefore, semantically akin to non-structured data types.

A signal is a classifier used to specify data passed in an asynchronous communication. The structural semantics of signals are essentially the same as for structured data types, so signal instances are also kinds of compound values (see Figure 8.12). (For the behavioral semantics of asynchronous communication using signals, see 8.10.)

Data Type Behaviors

The UML 2 Specification allows data types to own operations, as well as classes. However, data types are not behaviored classifiers, so they cannot own behaviors to be used as methods for their operations. Since fUML requires that every non-abstract operation have a method, it would thus only be possible to have abstract operations on data types, which would not be very useful. Therefore, data types are prohibited from having operations at all in fUML (see the constraint in 7.6.1).

It is thus not possible to use owned operations to define the primitive behaviors of a data type. Instead, the Foundational Model Library defines a set of primitive function behaviors that take values of primitive data types as their arguments. Rather than being operations of the primitive types, these primitive behaviors are grouped into library packages corresponding to the appropriate types (e.g., IntegerFunctions for type Integer, etc.). Implementations for these behaviors are then registered with the execution factory as part of the configuration of the execution environment (see 8.3.1).

Not being operations, such primitive behaviors are, of course, not polymorphic (see 7.7.1 on the semantics of polymorphic operation dispatching). They are called using call behavior actions, rather than call operation actions.

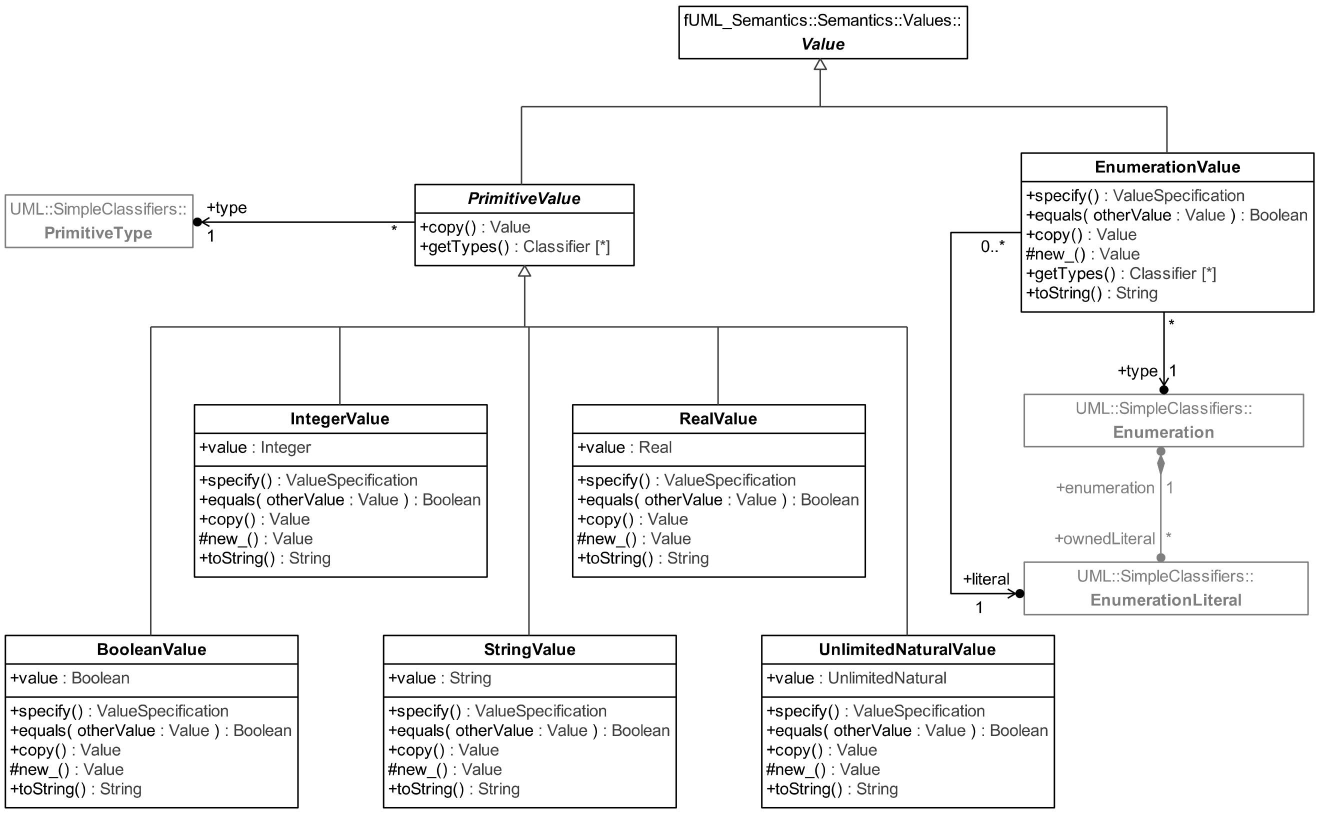


Figure 8.11 - Simple Values

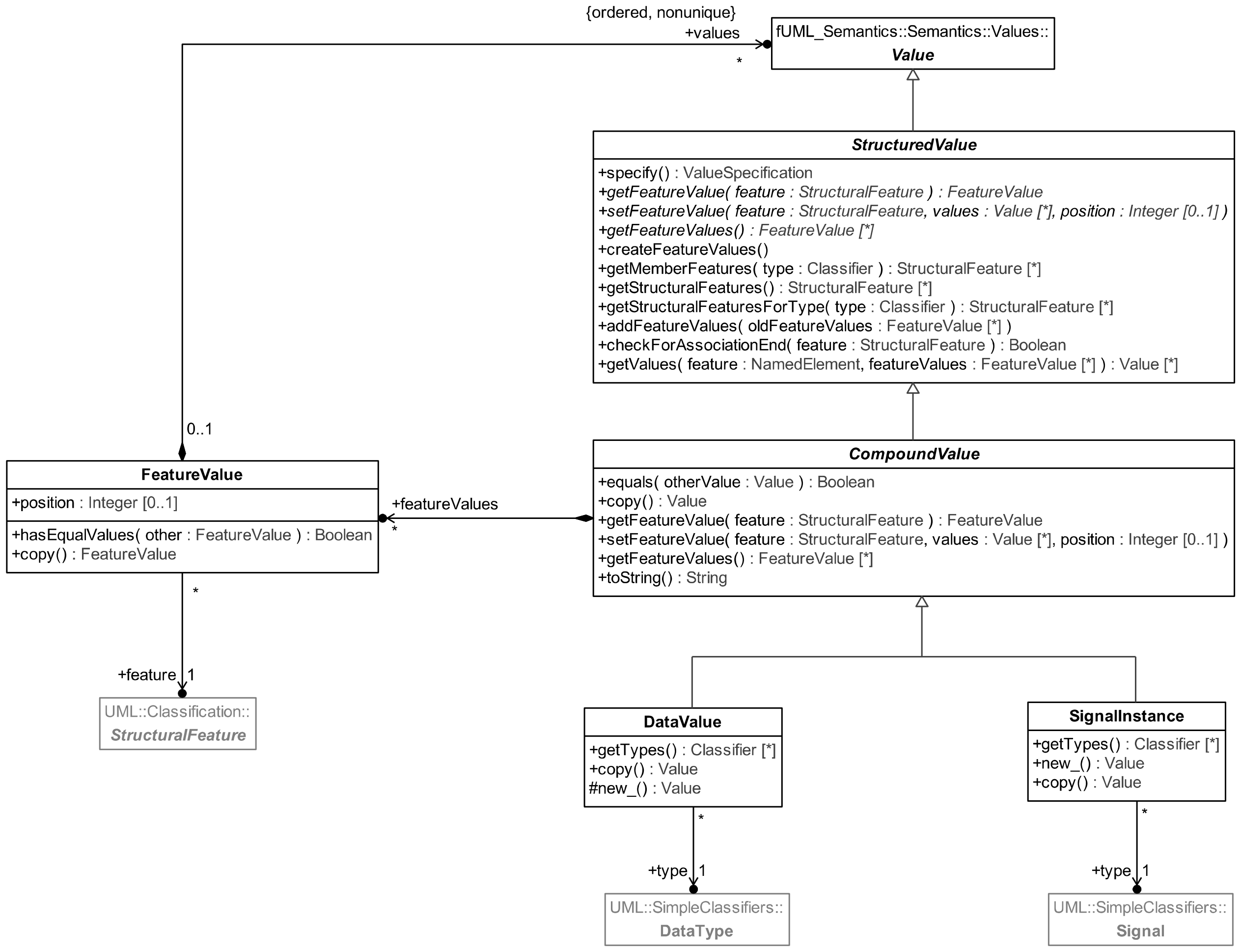

{ordered, nonunique}
+values
*
fUML_Semantics::Semantics::Values::
Value
StructuredValue
+specify() : ValueSpecification
+getFeatureValue(feature : StructuralFeature) : FeatureValue
+setFeatureValue(feature : StructuralFeature, values : Value [*], position : Integer [0..1])
+getFeatureValues() : FeatureValue [*]
+createFeatureValues()
+getMemberFeatures(type : Classifier) : StructuralFeature [*]
+getStructuralFeatures() : StructuralFeature [*]
+getStructuralFeaturesForType(type : Classifier) : StructuralFeature [*]
+addFeatureValues(oldFeatureValues : FeatureValue [*])
+checkForAssociationEnd(feature : StructuralFeature) : Boolean
+getValues(feature : NamedElement, featureValues : FeatureValue [*]) : Value [*]
0..1
FeatureValue
+position : Integer [0..1]
+hasEqualValues(other : FeatureValue) : Boolean
+copy() : FeatureValue
+featureValues
*
CompoundValue
+equals(otherValue : Value) : Boolean
+copy() : Value
+getFeatureValue(feature : StructuralFeature) : FeatureValue
+setFeatureValue(feature : StructuralFeature, values : Value [*], position : Integer [0..1])
+getFeatureValues() : FeatureValue [*]
+toString() : String
*
+feature 1
UML::Classification::
StructuralFeature
DataValue
+getTypes() : Classifier [*]
+copy() : Value
#new_() : Value
SignalInstance
+getTypes() : Classifier [*]
+new_() : Value
+copy() : Value
*
+type 1
UML::SimpleClassifiers::
DataType
*
+type 1
UML::SimpleClassifiers::
Signal

Figure 8.12 - Compound Values

8.6.2 Class Descriptions

8.6.2.1 BooleanValue

A boolean value is a primitive value whose type is Boolean.

Generalizations

- PrimitiveValue

Attributes

- value : Boolean
  The actual Boolean value.

Associations

None

Operations

[1] copy ( ) : Value

```
// Create a new boolean value with the same value as this boolean value.

BooleanValue newValue = (BooleanValue)(super.copy());

newValue.value = this.value;
return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this boolean value is equal to the otherValue.
// To be equal, the otherValue must have the same value as this boolean value.

boolean isEqual = false;
if (otherValue instanceof BooleanValue) {
     isEqual = ((BooleanValue)otherValue).value == this.value;
}

return isEqual;
```

[3] new_ ( ) : Value

```
// Return a new boolean value with no value.

return new BooleanValue();
```

[4] specify ( ) : ValueSpecification

```
// Return a literal boolean with the value of this boolean value.

LiteralBoolean literal = new LiteralBoolean();

literal.type = this.type;
literal.value = this.value;

return literal;
```

[5] toString ( ) : String

```
String stringValue = "false";

if (this.value) {
  stringValue = "true";
}

return stringValue;
```

8.6.2.2 CompoundValue

A compound value is a structured value with by-value semantics. Values are associated with each structural feature specified by the type(s) of the compound value.

Generalizations

- StructuredValue

Attributes

None

Associations

- featureValues : FeatureValue [0..*]

Operations

[1] copy ( ) : Value

```
// Create a new data value with the same featureValues as this data value.

CompoundValue newValue = (CompoundValue)(super.copy());

FeatureValueList featureValues = this.featureValues;
for (int i = 0; i < featureValues.size(); i++) {
     FeatureValue featureValue = featureValues.getValue(i);
     newValue.featureValues.addValue(featureValue.copy());
}

return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this data value is equal to the otherValue.
// To be equal, the otherValue must also be a compound value with the same types and equal
values for each feature.

boolean isEqual = otherValue instanceof CompoundValue;

 if (isEqual) {

     CompoundValue otherCompoundValue = (CompoundValue)otherValue;
     isEqual = super.equals(otherValue) & otherCompoundValue.featureValues.size() ==
this.featureValues.size();
```

```
    int i = 1;
    while (isEqual & i <= this.featureValues.size()) {
        FeatureValue thisFeatureValue = this.featureValues.getValue(i-1);

        boolean matched = false;
        int j = 1;
        while (!matched & j <= otherCompoundValue.featureValues.size()) {
            FeatureValue otherFeatureValue = otherCompoundValue.featureValues.getValue(j-1);
            if (thisFeatureValue.feature == otherFeatureValue.feature) {
                matched = thisFeatureValue.hasEqualValues(otherFeatureValue);
            }
            j = j + 1;
        }

        isEqual = matched;
        i = i + 1;
    }
}

return isEqual;
```

[3] getFeatureValue (in feature : StructuralFeature) : FeatureValue

```
// Get the value(s) of the member of featureValues for the given feature.

FeatureValue featureValue = null;
int i = 1;
while (featureValue == null & i <= this.featureValues.size()) {
    if (this.featureValues.getValue(i-1).feature == feature) {
        featureValue = this.featureValues.getValue(i-1);
    }
    i = i + 1;
}

return featureValue;
```

[4] getFeatureValues ( ) : FeatureValue [0..*]

```
// Return the feature values for this compound value.

return this.featureValues;
```

[5] setFeatureValue (in feature : StructuralFeature, in values : Value [0..*], in position : Integer [0..1])

```
// Set the value(s) of the member of featureValues for the given feature.
```

```
FeatureValue featureValue = this.getFeatureValue(feature);

if (featureValue == null) {
     featureValue = new FeatureValue();
     this.featureValues.addValue(featureValue);
}

featureValue.feature = feature;
featureValue.values = values;
featureValue.position = position;
```

[6] toString ( ) : String

```
String buffer = "(";

ClassifierList types = this.getTypes();

int i = 1;
while (i <= types.size()) {
    if (i != 1) {
        buffer = buffer + " ";
    }
    buffer = buffer + types.getValue(i - 1).name;
    i = i + 1;
}

int k = 1;
while (k <= this.featureValues.size()) {
     FeatureValue featureValue = this.featureValues.getValue(k-1);
     buffer = buffer + "\n\t\t" +  featureValue.feature.name + "[" + featureValue.position +
"]  =";

    int j = 1;
     while (j <= featureValue.values.size()) {
        Value value = featureValue.values.getValue(j - 1);
        if (value instanceof Reference) {
          Object_ object = ((Reference)value).referent;
          buffer = buffer + " Reference to " + object.identifier + "(";
          types = object.getTypes();
          int n = 1;
          while (n <= types.size()) {
            if (n != 1) {
                buffer = buffer + " ";
            }
            buffer = buffer + types.getValue(n - 1).name;
            n = n + 1;
          }
          buffer = buffer + ")";
        } else {
          buffer = buffer + " " + value.toString();
        }
        j = j + 1;
    }

    k = k + 1;
}

return buffer + ")";
```

8.6.2.3 DataValue

A data value is a compound value whose (single) type is a data type other than a primitive type or an enumeration.

Generalizations

- CompoundValue

Attributes

None

Associations

- type : DataType
  The type of this data value. This must not be a primitive or an enumeration.

Operations

[1] copy ( ) : Value

```
// Create a new data value with the same type and feature values as this data value.

DataValue newValue = (DataValue)(super.copy());

newValue.type = this.type;

return newValue;
```

[2] getTypes ( ) : Classifier [0..*]

```
// Return the single type of this data value.

ClassifierList types = new ClassifierList();
types.addValue(this.type);

return types;
```

[3] new_ ( ) : Value

```
// Create a new data value with no type or feature values.

return new DataValue();
```

8.6.2.4 EnumerationValue

An enumeration value is a value whose (single) type is an enumeration.

Its literal must be an owned literal of its type.

Generalizations

- Value

Attributes

None

Associations

- literal : EnumerationLiteral
  The literal value of this enumeration value.
- type : Enumeration

Operations

[1] copy ( ) : Value

```
// Create a new enumeration value with the same literal as this enumeration value.

EnumerationValue newValue = (EnumerationValue)(super.copy());

newValue.type = this.type;
newValue.literal = this.literal;

return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this enumeration value is equal to the otherValue.
// To be equal, the otherValue must also be an enumeration value with the same literal as
this enumeration value.

boolean isEqual = false;
if (otherValue instanceof EnumerationValue) {
     isEqual = ((EnumerationValue)otherValue).literal == this.literal;
}

return isEqual;
```

[3] getTypes ( ) : Classifier [0..*]

```
// Return the single type of this enumeration value.

ClassifierList types = new ClassifierList();
types.addValue(this.type);

return types;
```

[4] new_ ( ) : Value

```
// Create a new enumeration value with no literal.

return new EnumerationValue();
```

[5] specify ( ) : ValueSpecification

```
// Return an instance value with literal as the instance.

InstanceValue instanceValue = new InstanceValue();
InstanceSpecification instance = new InstanceSpecification();

instanceValue.type = this.type;
instanceValue.instance = this.literal;

return instanceValue;
```

[6] toString ( ) : String

```
return literal.name;
```

8.6.2.5 FeatureValue

A feature value gives the value(s) that a single structural feature has in a specific structured value.

Generalizations

None

Attributes

- position : Integer [0..1]
  The position of this feature value in a set of ordered values for a feature of an association. [This is only relevant if the feature value is for a link and the feature is ordered.]

Associations

- feature : StructuralFeature
  The structural feature being given value(s).
- values : Value [0..*]
  The values of for the feature. Zero or more values are possible, as constrained by the multiplicity of the feature.

Operations

[1] copy ( ) : FeatureValue

```
// Create a copy of this feature value.

FeatureValue newValue = new FeatureValue();

newValue.feature = this.feature;
newValue.position = this.position;

ValueList values = this.values;
for (int i = 0; i < values.size(); i ++) {
     Value value = values.getValue(i);
     newValue.values.addValue(value.copy());
}

return newValue;
```

[2] hasEqualValues (in other : FeatureValue) : Boolean

```
// Determine if this feature value has an equal set of values as another feature value.
// If the feature is ordered, then the values also have to be in the same order.

boolean equal = true;

if (this.values.size() != other.values.size()) {
     equal = false;

} else {
    if (this.feature.multiplicityElement.isOrdered) {
        int i = 1;
        while (equal & i <= this.values.size()) {
            equal = this.values.getValue(i-1).equals(other.values.getValue(i-1));
            i = i + 1;
        }

    } else {

        // Note: otherFeatureValues is used here solely as a holder for a copy of the list of
other values,
        // since the Java to UML mapping conventions do not allow "remove" on a local list
variable.
        FeatureValue otherFeatureValues = new FeatureValue();
        ValueList values = other.values;
        for (int i=0; i < values.size(); i++) {
            Value value = values.getValue(i);
            otherFeatureValues.values.addValue(value);
        }

        int i = 1;
        while (equal & i <= this.values.size()) {
            boolean  matched = false;
            int j = 1;
            while (!matched & j <= otherFeatureValues.values.size()) {
                if (this.values.getValue(i-1).equals(otherFeatureValues.values.getValue(j-
1))) {
                    matched = true;
                    otherFeatureValues.values.remove(j-1);

                }
                j = j + 1;
            }

            equal = matched;
            i = i + 1;
        }
    }
}
```

```
return equal;
```

8.6.2.6 IntegerValue

An integer value is a primitive value whose type is Integer.

Generalizations

- PrimitiveValue

Attributes

- value : Integer
  The actual Integer value.

Associations

None

Operations

[1] copy ( ) : Value

```
// Create a new integer value with the same value as this integer value.

IntegerValue newValue = (IntegerValue)(super.copy());

newValue.value = this.value;
return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this integer value is equal to the otherValue.
// To be equal, the otherValue must have the same value as this integer value.

boolean isEqual = false;
if (otherValue instanceof IntegerValue) {
     isEqual = ((IntegerValue)otherValue).value == this.value;
}

return isEqual;
```

[3] new_ ( ) : Value

```
// Create a new integer value with no value.

return new IntegerValue();
```

[4] specify ( ) : ValueSpecification

```
// Return a literal integer with the value of this integer value.

LiteralInteger literal = new LiteralInteger();

literal.type = this.type;
literal.value = this.value;

return literal;
```

[5] toString ( ) : String

```
String stringValue = "";

if (this.value == 0) {
    stringValue = "0";
} else {
    int positiveValue = this.value;

    if (positiveValue < 0) {
      positiveValue = -positiveValue;
    }

    do {
        int digit = positiveValue % 10;

        if (digit == 0) {
            stringValue = "0" + stringValue;
        } else if (digit == 1) {
            stringValue = "1" + stringValue;
        } else if (digit == 2) {
            stringValue = "2" + stringValue;
        } else if (digit == 3) {
            stringValue = "3" + stringValue;
        } else if (digit == 4) {
            stringValue = "4" + stringValue;
        } else if (digit == 5) {
            stringValue = "5" + stringValue;
        } else if (digit == 6) {
            stringValue = "6" + stringValue;
        } else if (digit == 7) {
            stringValue = "7" + stringValue;
        } else if (digit == 8) {
            stringValue = "8" + stringValue;
        } else if (digit == 9) {
            stringValue = "9" + stringValue;
        }

        positiveValue = positiveValue / 10;
    } while (positiveValue > 0);

    if (this.value < 0) {
        stringValue = "-" + stringValue;
    }
}

return stringValue;
```

8.6.2.7 PrimitiveValue

A primitive value is a value whose (single) type is a primitive type.

Generalizations

- Value

Attributes

None

Associations

- type : PrimitiveType

Operations

[1] copy ( ) : Value

```
// Create a new value that is equal to this primitive value.

PrimitiveValue newValue = (PrimitiveValue)(super.copy());

newValue.type = this.type;
return newValue;
```

[2] getTypes ( ) : Classifier [0..*]

```
// Return the single primitive type of this value.

ClassifierList types = new ClassifierList();
```

```
types.addValue(this.type);
return types;
```

8.6.2.8 RealValue

A real value is a primitive value whose type is real.

Generalizations

- PrimitiveValue

Attributes

- value : Real
  The actual Real value.

Associations

None

Operations

[1] copy ( ) : Value

```
// Create a new real value with the same value as this real value.

RealValue newValue = (RealValue)(super.copy());

newValue.value = this.value;
return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this real value is equal to the otherValue.
// To be equal, the otherValue must have the same value as this real value.

boolean isEqual = false;
if (otherValue instanceof RealValue) {
     isEqual = ((RealValue)otherValue).value == this.value;
}

return isEqual;
```

[3] new_ ( ) : Value

```
// Create a new real value with no value.
return new RealValue();
```

[4] specify ( ) : ValueSpecification

```
// Return a literal real with the value of this real value.

LiteralReal literal = new LiteralReal();

literal.type = this.type;
literal.value = this.value;

return literal;
```

[5] toString ( ) : String

```
String stringValue = "";

if (this.value == 0) {
     stringValue = "0";
} else {
     float positiveValue = this.value;

     if (positiveValue < 0) {
          positiveValue = -positiveValue;
     }

     int exponent = 0;

     if (positiveValue < .1) {
          while (positiveValue < .1) {
               positiveValue = positiveValue * 10;
               exponent = exponent - 1;
          }
     } else if (positiveValue >= 1) {
          while (positiveValue >= 1) {
               positiveValue = positiveValue / 10;
               exponent = exponent + 1;
          }
     }

     // This gives 9 significant digits in the mantissa.
     for (int i=0; i<9; i++) {
          positiveValue = positiveValue * 10;
     }
```

```
    IntegerValue integerValue = new IntegerValue ();
    integerValue.value = (int)positiveValue;
    stringValue = "0." + integerValue.toString();
    integerValue.value = exponent;
    StringValue = stringValue + "E" + integerValue.toString();

    if (this.value < 0) {
        stringValue = "-" + stringValue;
    }
}

return stringValue;
```

8.6.2.9 SignalInstance

Generalizations

- CompoundValue

Attributes

None

Associations

- type : Signal

Operations

[1] copy ( ) : Value

```
// Create a new signal instance with the same type and feature values as this signal
instance.

SignalInstance newValue = (SignalInstance)(super.copy());

newValue.type = this.type;

return newValue;
```

[2] getTypes ( ) : Classifier [0..*]

```
// Return the single type of this signal instance.

ClassifierList types = new ClassifierList();

types.addValue(this.type);
```

```
return types;
```

[3] new_ ( ) : Value

```
// Create a new signal instance with no type or feature values.

return new SignalInstance();
```

8.6.2.10 StringValue

A string value is a primitive value whose type is String.

Generalizations

- PrimitiveValue

Attributes

- value : String

Associations

None

Operations

[1] copy ( ) : Value

```
// Create a new string value with the same value as this string value.

StringValue newValue = (StringValue)(super.copy());

newValue.value = this.value;
return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this string value is equal to the otherValue.
// To be equal, the otherValue must have the same value as this string value.

boolean isEqual = false;
if (otherValue instanceof StringValue) {
     isEqual = ((StringValue)otherValue).value.equals(this.value);
}
return isEqual;
```

[3] new_ ( ) : Value

```
// Create a new string value with no value.

return new StringValue();
```

[4] specify ( ) : ValueSpecification

```
// Return a literal string with the value of this string value.

LiteralString literal = new LiteralString();

literal.type = this.type;
literal.value = this.value;

return literal;
```

[5] toString ( ) : String

```
return value;
```

8.6.2.11 StructuredValue

A structured value is a Value whose type has structural features: a data type (but not a primitive type or enumeration), a class or an association.

Generalizations

- Value

Attributes

None

Associations

None

Operations

[1] addFeatureValues (in oldFeatureValues : FeatureValue [0..*])

```
// Add feature values for all non-association-end structural features
// of the types of this structured value and all its supertypes
// (including private features that are not inherited). If a feature
// has an old feature value in the given list, then use that to
// initialize the values of the corresponding new feature value.
// Otherwise leave the values of the new feature value empty.

// Note: Any common features that appear twice in the list will simply
// have their values set multiple times to the same thing.
StructuralFeatureList features = this.getStructuralFeatures();
```

```
for (int i = 0; i < features.size(); i++) {
    StructuralFeature feature = features.getValue(i);
    if (!this.checkForAssociationEnd(feature)) {
        this.setFeatureValue(feature,
                this.getValues(feature, oldFeatureValues), 0);
    }
}
```

[2] checkForAssociationEnd (in feature : StructuralFeature) : Boolean

```
boolean isAssociationEnd = false;
if (feature instanceof Property) {
    isAssociationEnd = ((Property)feature).association != null;
}
return isAssociationEnd;
```

[3] createFeatureValues ( )

```
// Create empty feature values for all non-association-end structural
// features of the types of this structured value and all its supertypes
// (including private features that are not inherited).

this.addFeatureValues(new FeatureValueList());
```

[4] getFeatureValue (in feature : StructuralFeature) : FeatureValue

```
Get the feature value associated with the given feature.
The given feature must be a structural feature of the type of the structured value.
```

[5] getFeatureValues ( ) : FeatureValue [0..*]

```
Return the feature values associated with this structural value.
```

[6] getMemberFeatures (in type : Classifier) : StructuralFeature [0..*]

```
// Return the features for this structured value that are members of the
// given type. (That is, they are owned or inherited by the given type,
// excluding private features of supertypes that are not inherited.)

StructuralFeatureList features = this.getStructuralFeatures();
StructuralFeatureList memberFeatures = new StructuralFeatureList();

if (type != null) {
    NamedElementList members = type.member;
    for (int i = 0; i < features.size(); i++) {
        StructuralFeature feature = features.getValue(i);
        Boolean isMember = false;
        int k = 1;
        while (k <= members.size() & !isMember) {
            NamedElement member = members.getValue(k-1);
            isMember = feature == member;
            k = k + 1;
        }
        if (isMember) {
            memberFeatures.addValue(feature);
        }
    }
}

return memberFeatures;
```

[7] getStructuralFeatures ( ) : StructuralFeature [0..*]

```
// Get all structural features of the types of this structured
// value and all of their supertypes (including private features
// that are not inherited).

StructuralFeatureList features = new StructuralFeatureList();
ClassifierList types = this.getTypes();

for (int i = 0; i < types.size(); i++) {
    Classifier type = types.getValue(i);
    StructuralFeatureList typeFeatures = this.getStructuralFeaturesForType(type);
    for (int j = 0; j < typeFeatures.size(); j++) {
        NamedElement supertypeFeature = typeFeatures.getValue(j);
        features.addValue((StructuralFeature)supertypeFeature);
    }
}

return features;
```

[8] getStructuralFeatureForType(in type : Classifier) : StructuralFeature [0..*]

```
// Get all structural features of the given type and all of its
// supertypes (including private features that are not inherited).
```

```
StructuralFeatureList features = new StructuralFeatureList();

// Get feature values for the owned structural features of the given type.
NamedElementList ownedMembers = type.ownedMember;
for (int j = 0; j < ownedMembers.size(); j++) {
    NamedElement ownedMember = ownedMembers.getValue(j);
    if (ownedMember instanceof StructuralFeature) {
        features.addValue((StructuralFeature)ownedMember);
    }
}

// Add features for the structural features of the supertypes
// of the given type. (Note that the features for supertypes
// always come after the owned features.)
ClassifierList supertypes = type.general;
for (int i = 0; i < supertypes.size(); i++) {
    Classifier supertype = supertypes.getValue(i);
    StructuralFeatureList supertypeFeatures = this.getStructuralFeaturesForType(supertype);
    for (int j = 0; j < supertypeFeatures.size(); j++) {
        NamedElement supertypeFeature = supertypeFeatures.getValue(j);
        features.addValue((StructuralFeature)supertypeFeature);
    }
}

return features;
```

[8] getValues(in feature : NamedElement, featureValues : FeatureValue [0..*]) : Value [0..*]

```
// Return the values from the feature value in the given list for the
// given feature. If there is no such feature value, return an empty
// list.

FeatureValue foundFeatureValue = null;

int i = 1;
while (foundFeatureValue == null & i <= featureValues.size()) {
     FeatureValue featureValue = featureValues.getValue(i-1);
     if (featureValue.feature == feature) {
          foundFeatureValue = featureValue;
     }
     i = i + 1;
}

ValueList values;
if (foundFeatureValue == null) {
     values = new ValueList();
} else {
     values = foundFeatureValue.values;
}

return values;
```

[9] setFeatureValue (in feature : StructuralFeature, in values : Value [0..*], in position : Integer [0..1])

```
Set the value(s) and, optionally, the position index associated with the given feature.
The given feature must be a structural feature of the type of the structured value.
```

[10] specify ( ) : ValueSpecification

```
// Return an instance value that specifies this structured value.

InstanceValue instanceValue = new InstanceValue();
InstanceSpecification instance = new InstanceSpecification();

instanceValue.type = null;
instanceValue.instance = instance;

instance.classifier = this.getTypes();

FeatureValueList featureValues = this.getFeatureValues();
for (int i = 0; i < featureValues.size(); i++) {
     FeatureValue featureValue = featureValues.getValue(i);

     Slot slot = new Slot();
     slot.definingFeature = featureValue.feature;

     ValueList values = featureValue.values;
     for (int j = 0; j < values.size(); j++) {
         Value value = values.getValue(j);
         slot.value.addValue(value.specify());
     }

     instance.slot.addValue(slot);
}

return instanceValue;
```

8.6.2.12 UnlimitedNaturalValue

An unlimited natural value is a primitive value whose type is UnlimitedNatural.

Generalizations

- PrimitiveValue

Attributes

- value : UnlimitedNatural
  The actual unlimited natural value.

Associations

None

Operations

[1] copy ( ) : Value

```
// Create a new unlimited natural value with the same value as this value.

UnlimitedNaturalValue newValue = (UnlimitedNaturalValue)(super.copy());

newValue.value = this.value;
return newValue;
```

[2] equals (in otherValue : Value) : Boolean

```
// Test if this unlimited natural value is equal to the otherValue.
// To be equal, the otherValue must have the same value as this unlimited natural value.

boolean isEqual = false;
if (otherValue instanceof UnlimitedNaturalValue) {
     isEqual = ((UnlimitedNaturalValue)otherValue).value.naturalValue ==
this.value.naturalValue;
}

return isEqual;
```

[3] new_ ( ) : Value

```
// Create a new unlimited natural value with no value.
```

```
return new UnlimitedNaturalValue();
```

[4] specify ( ) : ValueSpecification

```
// Return a literal unlimited natural with the value of this unlimited natural value.

LiteralUnlimitedNatural literal = new LiteralUnlimitedNatural();

literal.type = this.type;
literal.value = this.value;

return literal;
```

[5] toString ( ) : String

```
String stringValue = "*";

if (this.value.naturalValue >= 0) {
     IntegerValue integerValue = new IntegerValue();
     integerValue.value = this.value.naturalValue;
     stringValue = integerValue.toString();
}

return stringValue;
```

8.7 Structured Classifiers

8.7.1 Overview

Extensional Values

Every classifier has an intension, that is, the set of all possible values that may have that classifier as a type. Other than for enumerations, for which this set is explicitly specified, the intension of a classifier is conceptually infinite (though, of course, actually finite in any real implementation). In fact, one semantic mapping for a classifier is to have it specifically denote its intension.

However, there is a fundamental difference between the intensions of data types and classes. As discussed in 7.6.1, the possible values of a data type are fully determined by the definition of the type.On the other hand, an instance of a class, called an object, has an identity separate from the values of its attributes. Two objects can have the same values for their attributes, and still be distinct objects. Further, the values of the attributes of an object may change over time, independently of how the attribute values of any other object change.

Actually, an instance value of a class does not map directly to an object but, rather, to a reference to an object, as shown in Figure 8.13. This is because an object, once created, has an independent existence and there may be multiple references to that same object. Changes to the object made via one reference are visible via any other reference.

Objects are thus examples of extensional values, as are links, which are instances of associations. In addition to their intension, classes and associations have an extension, that is, the set of instances of the class or association that exist at any one point in time. This leads, however, to the issue of managing the scope of such extension sets.

This is particularly important for associations. There are actually no actions that return links as values. (Foundational UML does not contain association actions, so it does not provide semantics for link objects.) Rather, a read link action actually queries the current extension of the association for matching links.

But, pragmatically, how does one bound what is to be included in the actual extension set? Certainly links created during the execution of a model should be accessible later in the execution of that model. But what about other executions of the same model, perhaps widely physically distributed? What about other models that may reuse the same association?

In order to deal with this issue, the fUML semantic model introduces the concept of a locus, as shown in Figure 8.13 and described in 8.3. An existential value is created at a specific such locus and remains there during its life. The extent of a class or association is its extension at a specific locus.

For executions at a certain locus, the extension of a class or association is always limited to the extent at that locus. Therefore, a read link action will only query the specified association extent at the locus at which it is executing. Similarly, a read extent action will only return (references to) the set of currently extant objects in the specified class extent at the locus at which it is executing.

Polymorphic Operation Dispatching

Operations in UML are potentially polymorphic-that is, there may be multiple methods for any one operation. Polymorphic operation dispatching is the determination of which method to use for a given invocation of the operation, depending on the context and target of the invocation. The specification for this determination is provided in the execution model by the dispatch operation of the Object class, as shown in Figure 8.13 (the semantics of operation dispatching is further discussed in relation to the call operation action in 8.10).

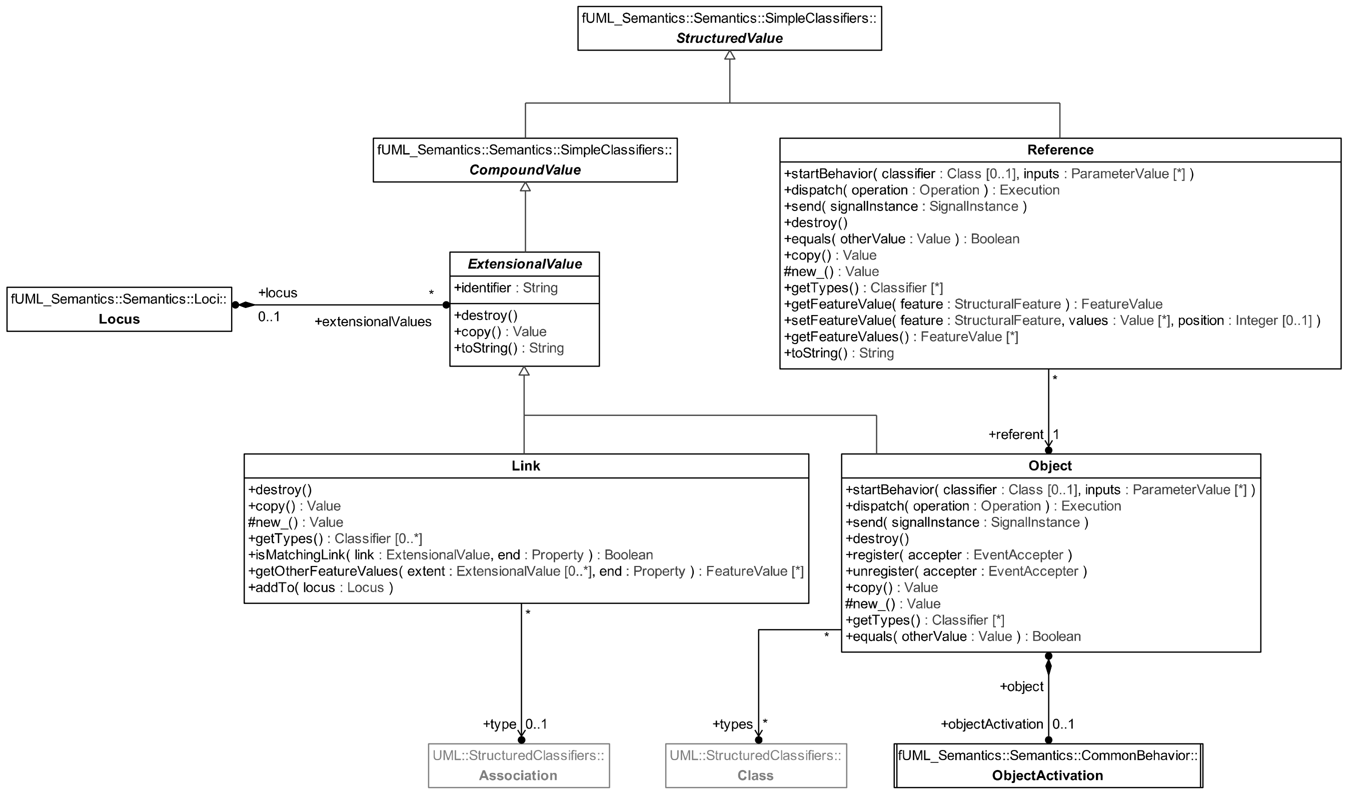
However, the exact behavior to be specified for polymorphic operation dispatching is a semantic variation point in fUML. (See 2.3 for a full discussion of semantic variation within fUML.) Following the general approach of using the Strategy Pattern to model semantic variation points (see 8.3.1), the variability of operation dispatching is captured by using strategy classes for the Object::dispatch operation. DispatchStrategy provides the abstract base class for this type of strategy (see Figure 8.14). The default dispatching behavior is given by the concrete class RedefinitionBasedDispatchStrategy.

The default redefinition based dispatch strategy requires that every concrete fUML operation has an associated method. In order to override an operation inherited from a superclass, the subclass must declare the redefining operation as a redefinition of the inherited operation. This is interpreted as meaning that any calls made to the original superclass operation, for objects that are instances of the subclass or any of its descendants, will be dispatched to the method of the redefining operation, rather than to the method of the original operation.

A conforming execution tool may define an alternative rule for how this dispatching is to take place by defining a new DispatchStrategy subclass specifying whatever rule is desired. An instance of this alternate strategy must then be registered with the execution factory at a given locus, rather than the default strategy.

To simplify the specification of new concrete dispatch strategy subclasses, the abstract base DispatchStrategy class provides a generally applicable method for its dispatch operation using a second operation, getMethod. The getMethod operation takes the same arguments as dispatch (the target object and the operation to be dispatched) and is required to return the operation method chosen to be executed for the operation by a specific dispatch strategy. The dispatch operation then creates an execution for the chosen method at the locus of the target object on which the operation is being invoked and returns that execution object.

It is also possible for a concrete operation to have no method, if calls to it are to be handled using a call event (see the discussion of accept call actions in 8.10). This case is still managed through the dispatch strategy. Instead of using a method behavior defined for the operation in the model, an instance of a special CallEventBehavior class (see 8.8) is created to act as an effective method for the call. For convenience, the getMethod operation in the DispatchStrategy superclass provides this functionality, which may be used as appropriate in the definition of the getMethod operation for a subclass of DispatchStrategy (for example, the redefinition based dispatch strategy getMethod operation calls the superclass operation if it identifies a concrete, most-redefined operation, but that operation does not have a method). The execution created for a call event behavior is an instance of the CallEventExecution class, which carries out the behavior of sending a call event occurrence and waiting for a response.



fUML_Semantics::Semantics::SimpleClassifiers::
StructuredValue
fUML_Semantics::Semantics::SimpleClassifiers::
CompoundValue
Reference
+startBehavior(classifier : Class [0..1], inputs : ParameterValue [*])
+dispatch(operation : Operation) : Execution
+send(signalInstance : SignalInstance)
+destroy()
+equals(otherValue : Value) : Boolean
+copy() : Value
#new_() : Value
+getTypes() : Classifier [*]
+getFeatureValue(feature : StructuralFeature) : FeatureValue
+setFeatureValue(feature : StructuralFeature, values : Value [*], position : Integer [0..1])
+getFeatureValues() : FeatureValue [*]
+toString() : String
ExtensionalValue
+identifier : String
+destroy()
+copy() : Value
+toString() : String
fUML_Semantics::Semantics::Loci::
Locus
+locus
0..1
*
+extensionalValues
*
+referent
1
Link
+destroy()
+copy() : Value
#new_() : Value
+getTypes() : Classifier [0..*]
+isMatchingLink(link : ExtensionalValue, end : Property) : Boolean
+getOtherFeatureValues(extent : ExtensionalValue [0..*], end : Property) : FeatureValue [*]
+addTo(locus : Locus)
Object
+startBehavior(classifier : Class [0..1], inputs : ParameterValue [*])
+dispatch(operation : Operation) : Execution
+send(signalInstance : SignalInstance)
+destroy()
+register(accepter : EventAccepter)
+unregister(accepter : EventAccepter)
+copy() : Value
#new_() : Value
+getTypes() : Classifier [*]
+equals(otherValue : Value) : Boolean
*
*
+object
+type
0..1
+types
*
+objectActivation
0..1
UML::StructuredClassifiers::
Association
UML::StructuredClassifiers::
Class
fUML_Semantics::Semantics::CommonBehavior::
ObjectActivation

Figure 8.13 - Extensional Values

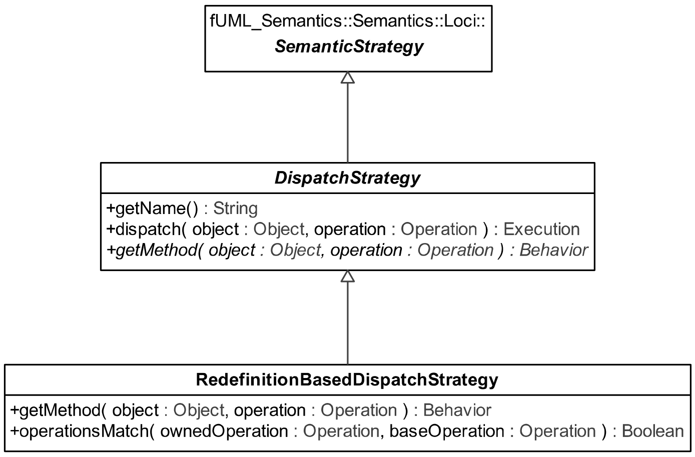

fUML_Semantics::Semantics::Loci::
SemanticStrategy
DispatchStrategy
+getName() : String
+dispatch(object : Object, operation : Operation) : Execution
+getMethod(object : Object, operation : Operation) : Behavior
RedefinitionBasedDispatchStrategy
+getMethod(object : Object, operation : Operation) : Behavior
+operationsMatch(ownedOperation : Operation, baseOperation : Operation) : Boolean

Figure 8.14 - Dispatch Strategies

8.7.2 Class Descriptions

8.7.2.1 DispatchStrategy

A dispatch strategy is a semantic strategy for the polymorphic dispatching of an operation to an execution of a method for that operation.

Generalizations

- SemanticStrategy

Attributes

None

Associations

None

Operations

[1] dispatch (in object : Object, in operation : Operation) : Execution

```
// Get the behavior for the given operation as determined by the type(s) of the given object,
compile the behavior at the locus of the object, and return the resulting execution object.

return  object.locus.factory.createExecution(this.getMethod(object,operation),  object);
```

[2] getMethod (in object : Object, in operation : Operation) : Behavior

```
// Get the method that corresponds to the given operation for the given object.
// By default, the operation is treated as being called via a call event occurrence,
// with a call event behavior as its effective method. Concrete dispatch strategy
// subclasses may override this default to provide other dispatching behavior.

CallEventBehavior method = new CallEventBehavior();
method.setOperation(operation);
return method;
```

[3] getName ( ) : String

```
// Dispatch strategies are always named "dispatch".

return "dispatch";
```

8.7.2.2 ExtensionalValue

An extensional value is a data value that is part of the extent of some classifier at a specific locus.

Generalizations

- CompoundValue

Attributes

- identifier : String
  The identifier for this extensional value, unique among the extensional values created at the same locus as this value.

Associations

- locus : Locus [0..1]
  The locus of the extent of which this value is a member. (If the value has been destroyed, it has no locus.)

Operations

[1] copy ( ) : Value

```
// Create a new extensional value with the same feature values at the same locus as this one.

ExtensionalValue newValue = (ExtensionalValue)(super.copy());

if (this.locus != null) {
     this.locus.add(newValue);
}

return newValue;
```

[2] destroy ( )

```
// Remove this value from its locus (if it has not already been destroyed).
```

```
if (this.locus != null) {
     this.locus.remove(this);
}
```

[3] toString ( ) : String

```
return this.identifier + super.toString();
```

8.7.2.3 Link

A link is an extensional value whose (single) type is an association. (However, if the link has been destroyed, then it has no type.)

A link must at have most one feature value for each structural feature owned by its type.

Generalizations

- ExtensionalValue

Attributes

None

Associations

- type : Association [0..1]
  The type of this link.

Operations

```
// Return a literal integer with the value of this integer value.

LiteralInteger literal = new LiteralInteger();

literal.type = this.type;
```

[1] addTo (locus : Locus)

```
// Add this link to the extent of its association at the given locus,
// Shift the positions of ends of other links, as appropriate, for ends
// that are ordered.

PropertyList ends = this.type.memberEnd;
ExtensionalValueList extent = locus.getExtent (this.type);

for (int i = 0; i < ends.size(); i++) {
     Property end = ends.getValue(i);
     if (end.multiplicityElement.isOrdered) {
          FeatureValue featureValue = this.getFeatureValue(end);
          FeatureValueList otherFeatureValues =
                this.getOtherFeatureValues(extent, end);
            int n = otherFeatureValues.size();
            if (featureValue.position < 0 | featureValue.position > n) {
                featureValue.position = n + 1;
            } else {
                if (featureValue.position == 0) {
                    featureValue.position - 1;
                }
                for (int j = 0; j < otherFeatureValues.size(); j++) {
                    FeatureValue otherFeatureValue = otherFeatureValues.getValue(j);
                    if (featureValue.position <= otherFeatureValue.position) {
                        otherFeatureValue.position = otherFeatureValue.position + 1;
                    }
                }
            }
        }
    }

locus.add(this);
```

[2] copy ( ) : Value

```
// Create a new link with the same type, locus and feature values as this link.

Link newValue = (Link)(super.copy());

newValue.type = this.type;

return newValue;
```

[3] destroy ( )

```
// Remove the type of this link and destroy it.
// Shift the positions of the feature values of any remaining links in
// the extent of the same association, for ends that are ordered.

PropertyList ends = this.type.memberEnd;
ExtensionalValueList extent = this.locus.getExtent(this.type);

for (int i = 0; i < extent.size(); i++) {
     ExtensionalValue otherLink = extent.getValue(i);
     for (int j=0; j < ends.size(); j++) {
         Property end = ends.getValue(j);
         if (end.multiplicityElement.isOrdered) {
             FeatureValue featureValue = otherLink.getFeatureValue(end);
             if (this.getFeatureValue(end).position < featureValue.position) {
                 featureValue.position = featureValue.position - 1;
             }
         }
     }
}

this.type = null;
super.destroy();
```

[4] getOtherFeatureValues (extent : ExtensionalValue [*]. end : Property) : FeatureValue [*]

```
// Return all feature values for the given end of links in the given
// extent whose other ends match this link.

FeatureValueList_featureValues = new FeatureValueList();
for (int i = 0; i < extent.size(); i++) {
     ExtensionValue link = extent.getValue(i);
     if (link != this {
          if (isMatchingLink(link, end)) {
               featureValues.addValue(link.getFeatureValue(end));
          }
     }
}
return  featureValues;
```

[5] getTypes ( ) : Classifier [0..*]

```
// Return the single type of this link (if any).

ClassifierList types = null;

if (this.type == null) {
     types = new ClassifierList();
} else {
     types = new ClassifierList();
     types.addValue(this.type);
}
```

```
return types;
```

[6] isMatchingLink (link : ExtensionalValue. end : Property) : Boolean

```
// Test whether the given link matches the values of this link on all
// ends other than the given end.

PropertyList ends = this.type.memberEnd;

boolean matches = true;
int i = 1;
while (matches & i <= ends.size()) {
     Property otherEnd = ends.getValue(i - 1);
     if (otherEnd != end &
               !this.getFeatureValue(otherEnd).values.getValue(0).equals(
               link.getFeatureValue(otherEnd).values.getValue(0))) {
          matches = false;
     }
     i = i + 1;
}

return matches;
```

[7] new_ ( ) : Value

```
// Create a new link with no type or properties.

return new Link();
```

8.7.2.4 Object

An object is an extensional value that may have multiple types, all of which must be classes. (Note that a destroyed object has no types.)

An object has a unique identity. Usually, references to objects are manipulated, rather than the objects themselves, and there may be multiple references to the same object.

If an object is active, it has an object activation that handles the execution of its classifier behavior(s).

Generalizations

- ExtensionalValue

Attributes

None

Associations

- objectActivation : ObjectActivation
  The object activation handling the active behavior of this object.
- types : Class
  The classes under which this object is currently classified. (A destroyed object has no types.)

Operations

[1] copy ( ) : Value

```
// Create a new object that is a copy of this object at the same locus as this object.
// However, the new object will NOT have any object activation (i.e, its classifier behaviors
will not be started).

Object_ newObject = (Object_)(super.copy());

Class_List types = this.types;
for (int i = 0; i < types.size(); i++) {
     Class_ type = types.getValue(i);
     newObject.types.addValue(type);
}

return newObject;
```

[2] destroy ( )

```
// Stop the object activation (if any), clear all types, clear all feature values
// and destroy the object as an extensional value.

if (this.objectActivation != null) {
     this.objectActivation.stop();
     this.objectActivation = null;
}

this.types.clear();
this.featureValues.clear();
super.destroy();
```

[3] dispatch (in operation : Operation) : Execution

```
// Dispatch the given operation to a method execution, using a dispatch strategy.

return  ((DispatchStrategy)this.locus.factory.getStrategy("dispatch")).dispatch(this,
operation);
```

[4] equals (in otherValue : Value) : Boolean

```
// Test if this object is equal to the otherValue.
// To be equal, the otherValue must be the same object as this object.

return this == otherValue;
```

[4] getTypes ( ) : Classifier [0..*]

```
// Return the types of this object.

ClassifierList types = new ClassifierList();
Class_List myTypes = this.types;
for (int i = 0; i < myTypes.size(); i++) {
     Class_ type = myTypes.getValue(i);
     types.addValue(type);
}

return types;
```

[5] new_ ( ) : Value

```
// Create a new object with no type, feature values or locus.

return new Object_();
```

[6] register (in accepter : EventAccepter)

```
// Register the given accept event accepter to wait for a dispatched signal event.

if (this.objectActivation != null) {
     this.objectActivation.register(accepter);
}
```

[7] send (in eventOccurrence : EventOccurrence)

```
// If the object is active, add the given event occurrence to the event
// pool and signal that a new event occurrence has arrived.

if (this.objectActivation != null) {
     this.objectActivation.send(eventOccurrence);
}
```

[8] startBehavior (in classifier : Class [0..1], in inputs : ParameterValue [0..*])

```
// Create an object activation for this object (if one does not already exist) and start its
behavior(s).

if (this.objectActivation == null) {
     this.objectActivation = new ObjectActivation();
     this.objectActivation.object = this;
}

this.objectActivation.startBehavior(classifier,  inputs);
```

[9] unregister (in accepter : EventAccepter)

```
// Remove the given event accepter for the list of waiting event accepters.

if (this.objectActivation != null) {
     this.objectActivation.unregister(accepter);
}
```

8.7.2.5 RedefinitionBasedDispatchStrategy

A redefinition-based dispatch strategy is one that requires an overriding subclass operation to explicitly redefine the overridden superclass operation. If a concrete operation has no methods, then it is assumed to be handled by a call event; otherwise, it should have at most one method.

Generalizations

- DispatchStrategy

Attributes

None

Associations

None

Operations

[1] getMethod (in object : Object, in operation : Operation) : Behavior

```
// Find the member operation of a type of the given object that
// is the same as or a redefinition of the given operation. Then
// return the method of that operation, if it has one, otherwise
// return a CallEventBehavior as the effective method for the
// matching operation.
// [If there is more than one type with a matching operation, then
// the first one is arbitrarily chosen.]

Behavior method = null;
```

```
int i = 1;
while (method == null & i <= object.types.size()) {
     Class_ type = object.types.getValue(i-1);
     NamedElementList members = type.member;
     int j = 1;
     while (method == null & j <= members.size()) {
          NamedElement member = members.getValue(j-1);
          if (member instanceof Operation) {
               Operation memberOperation = (Operation)member;
               if (this.operationsMatch(memberOperation, operation)) {
                    if (memberOperation.method.size() == 0) {
                         method = super.getMethod(object, memberOperation);
                    } else {
                         method = memberOperation.method.getValue(0);
                    }
               }
          }
          j = j + 1;
     }
     i = i + 1;
}

return method;
```

8.7.2.6 Reference

A reference is an access path to a specific object. There may be multiple references to the same object.

As a structured value, the reference acts just the same as its referent in terms of type, features, operations, etc.

Generalizations

- StructuredValue

Attributes

None

Associations

- referent : Object

Operations

[1] copy ( ) : Value

```
// Create a new reference with the same referent as this reference.
```

```
Reference newValue = (Reference)(super.copy());

newValue.referent = this.referent;

return newValue;
```

[2] destroy ( )

```
// Destroy the referent.

this.referent.destroy();
```

[3] dispatch (in operation : Operation) : Execution

```
// Dispatch the given operation to the referent object.

return  this.referent.dispatch(operation);
```

[4] equals (in otherValue : Value) : Boolean

```
// Test if this reference is equal to the otherValue.
// To be equal, the otherValue must also be a reference, with the same referent as this
reference.

boolean isEqual = false;
if (otherValue instanceof Reference) {
     if (this.referent == null) {
          isEqual = ((Reference)otherValue).referent == null;
     } else {
          isEqual = this.referent.equals(((Reference) otherValue).referent);
     }
}

return isEqual;
```

[5] getFeatureValue (in feature : StructuralFeature) : FeatureValue

```
// Get the feature value associated with the given feature in the referent object.

return  this.referent.getFeatureValue(feature);
```

[6] getFeatureValues ( ) : FeatureValue [0..*]

```
// Return the feature values of the referent.
```

```
return  this.referent.getFeatureValues();
```

[7] getTypes ( ) : Classifier [0..*]

```
// Get the types of the referent object.

return  this.referent.getTypes();
```

[8] new_ ( ) : Value

```
// Create a new reference with no referent.

return new Reference();
```

[9] send (in eventOccurrence : EventOccurrence)

```
// Send the given signal instance to the referent object.

this.referent.send(eventOccurrence);
```

[10] setFeatureValue (in feature : StructuralFeature, in values : Value [0..*], in position : Integer [0..1])

```
// Set the values associated with the given feature in the referent object.

this.referent.setFeatureValue(feature,  values,  position);
```

[11] startBehavior (in classifier : Class [0..1], in inputs : ParameterValue [0..*])

```
// Asynchronously start the behavior of the given classifier for the referent object.

this.referent.startBehavior(classifier,  inputs);
```

[12] toString ( ) : String

```
return "Reference to " + this.referent.toString();
```

8.8 Common Behavior

8.8.1 Overview

Executions

In UML, a behavior is actually a kind of class, and it may, therefore, have instances. An instance of a behavior is called an execution, as shown in Figure 8.15. An instance value with a behavior type thus evaluates to an execution object.

The abstract Execution class has two concrete subclasses: OpaqueBehaviorExecution (shown in Figure 8.15) and ActivityExecution (see 8.9). These subclasses act as visitor classes for OpaqueBehavior and Activity, respectively (see 8.3 for a general discussion of visitor classes). (Since function behaviors are basically just opaque behaviors with certain additional restrictions, OpaqueBehaviorExecution also acts as the visitor class for FunctionBehavior.)

To execute a behavior, the executor uses the execution factory to create an instance of the appropriate execution class (see 8.3). The behavior to be executed becomes the type of the instantiated execution object. The executor then sets the parameter values for the input parameters (i.e., those with direction in and in-out) of the behavior (if any) and calls the execute operation on the execution object.

The Execution::execute operation provides the fundamental specification of behavior in fUML. The execute operation is actually defined as an abstract operation on the Execution class, since its detailed specification depends on the kind of behavior being executed. See 8.9 for a specific discussion of the execution of activities, which provide the means for user modeling of behavior in fUML.

In general, though, the execute operation of an execution must act on the parameter values for any input parameters (i.e., those with direction in or inout) and produce parameter values for any output parameters (i.e, those with direction inout or out). For normal parameters (that is, those with isStream = false), input parameter values are available before the execute operation is called and output parameter values are created at the end of the execution. Streaming parameters (that is, those with isStream = true), however, can accept values (for input parameters) or post values (for output parameters) while a behavior is executing.

To allow streaming, a streaming parameter *listener* can register to receive values posted to a streaming parameter during the execution of a behavior. For an input parameter, the listener receives input values posted after the execution begins and passes them to some element within the behavior execution (for example, an input parameter activity node for an activity execution, see 8.9.2.7). For an output parameter, the listener receives output values posted from within the behavior execution to the invoker of the behavior (for example, an output pin of a call action invoking a behavior with streaming parameters, see 8.10.2.26).

Note that, as a kind of object itself, an execution is an extensional value. As discussed in 8.7, this means that any execution effectively takes place at a specific locus. Thus, an object created during an execution will exist at the locus of the execution. Unless this new object is explicitly destroyed later in the execution, it will continue to exist in the extent of its class at the execution locus, even after the behavior that created it has completed its execution.

Active Objects

An active object is one that has one or more classifiers that are active classes – that is, they are classes with a classifier behavior. (In fUML, an active class must either be a behavior or have a classifier behavior and only active classes may be behaviored classifiers – see 7.9). After an active object is instantiated, a start object behavior action (see 7.11) is used to start one or more of its classifier behaviors. Note that an object may also become active if it has an active class added to it using a reclassify object action (see 7.11). In this case, a start object behavior action must still be used to start the classifier behavior of the newly added class.

Once started, classifier behaviors then run asynchronously from whatever behavior executed the start object behavior action. This allows the active object to autonomously send communications to and react to communications from other objects. The points at which an active object responds to asynchronous communications from other objects is determined solely by the behavior of the active object.

Active objects in fUML communicate asynchronously via signals. A signal is a kind of classifier (see 7.6). Therefore, an instance of a signal is a value. Since a signal may have attributes, a signal instance is a kind of compound value (see Figure 8.12).

The semantic model for an active object itself is an extension to the basic value model for objects. An active object still has the same structural semantics as a passive object, but it adds the behavioral semantics of the execution of classifier behaviors and the handling of asynchronous communication. These semantics are captured in the object activation for an active object, which is created when the active behavior of the object is started.

Note that a behavior may itself be instantiated as an active object and the active behavior of a behavior instance is just the behavior itself, acting as its own context object. For simplicity, in the description of the semantics of active objects and event handling below, the running behavior of an active object is always referred to as its "classifier behavior execution". However, in the case of an active object that is a behavior instance (which is already an execution), the "classifier behavior execution" will actually be the active object itself, not an instance of some other "classifier behavior".

Event Dispatching

An active object may asynchronously react to the occurrence of various events. When an object is notified of the occurrence of an event, it is said to have received the event occurrence. Asynchronicity means that the receipt of the event occurrence is decoupled from the dispatching of that occurrence, which is when a determination is made as to how the object will react to the event occurrence (if at all).

Note: Operation calls are always synchronous invocations in fUML, as opposed to signal sends, which are always asynchronous invocations. Nevertheless, when an operation call is handled by an active object via a call event, the call event occurrence is *handled* asynchronously by the object, in the sense discussed above. The synchronous nature of the call is maintained for the caller by requiring the caller to block the execution thread making the call until the call event occurrence is dispatched and replied to. See also the discussion of call event occurrences under Event Occurrences below.

In order to achieve this decoupling, ObjectActivation is itself an active class (in the execution model). The classifier behavior for ObjectActivation (see Figure 8.19) is a simple dispatch loop. When an event occurrence is received by an active object, it is placed into the event pool of the object activation for that object, after which the object activation sends an ArrivalSignal to itself. The dispatch loop waits for an ArrivalSignal and, when this happens, calls the dispatchNextEvent operation. This operation dispatches a single event occurrence from the event pool. Once this is complete ("run to completion semantics" for dispatched event occurrences), the dispatch loop returns to waiting for another event occurrence to arrive.

It is important to carefully note the two semantics levels in the above description. At the level of a user model, the execution model is modeling the receipt of an event occurrence and the dispatching of that event occurrence, to be handled as defined in the user model. However, the semantic model itself also uses the active class ObjectActivation and the signal ArrivalSignal, whose receipt by an object activation is an event occurrence handled by the classifier behavior of the object activation (i.e., the event dispatch loop). The semantics for active class and signals, as used in the execution model, are given by the base semantics for those model constructs (see Clause 10; also see Clause 6 for a general discussion of fUML execution semantics versus base semantics).

Note, while an event occurrence is being dispatched, it is possible that the active object will receive additional event occurrences. In this case, these event occurrences will be concurrently placed into the event pool for the active object and an ArrivalSignal will be generated for each arriving event occurrence. When the dispatch loop is ready to accept another event occurrence, it will accept exactly one pending ArrivalSignal, causing another event occurrence to be dispatched. The dispatch loop will continue to dispatch event occurrences, one at a time, until there are no more pending ArrivalSignals (or until the active object is destroyed).

Which event occurrence is actually dispatched out of the event pool is not determined by the ArrivalSignal but, rather, by the dispatchNextEvent operation. However, the exact behavior to be specified for this operation is a semantic variation point in fUML. (See 2.3 for a full discussion of semantic variation within fUML.)

Following the general approach of using the Strategy Pattern to model semantic variation points (see 8.3.1), the variability of event dispatching is captured by using strategy classes for the ObjectActivation::getNextEvent operation. GetNextEventStrategy provides the abstract base class for this type of strategy. The default dispatching behavior is given by the concrete FIFOGetNextEventStrategy, which dispatches event occurrences on a first-in first-out (FIFO queue) basis. Any variant behavior must be fully specified by overriding the behavioral specification of the dispatchNextEvent operation.

A conforming execution tool may define an alternative rule for how this dispatching is to take place by defining a new GetNextEventStrategy subclass specifying whatever rule is desired. An instance of this alternate strategy must then be registered with the execution factory at a given locus, rather than the default strategy.

Once an event occurrence is selected for dispatch, it is matched against the list of waiting event accepters for the active object. If a match is found, the event occurrence is passed to the event accepter using its accept operation. If no matching event acceptor is found, the event occurrence is not returned to the event pool and is lost. (Note that deferred events are not included in the fUML subset.)

The event accepters for an active object are points within the executing classifier behaviors of the object that are waiting for certain events. An executing classifier behavior may register an event accepter for itself using the Object::registerForEvent operation. The event accepter is then added to the list of waiting event accepters for the object and any matching event occurrence is passed back to the executing classifier behavior via the accept operation of the event accepter.

Event Occurrences

The event-dispatching framework described above is intended to be general enough to handle the occurrence of various different kinds of events defined in UML. However, currently there are three kinds of events whose occurrences are handled in the fUML execution model: classifier behavior (asynchronous) invocation events, signal reception events and operation call events. It is expected that other specifications building on fUML may specify the semantics of other kinds of events within the general fUML event-handling framework.

The EventOccurrence class is also an active class in the execution model, in a similar way to ObjectActivation, as described above. When an event occurrence is sent using the sendTo operation of EventOccurrence, the event occurrence classifier behavior is started asynchronously. When the classifier behavior executes, it carries out the actual sending of the event occurrence to the target object, resulting in the event occurrence being placed in the target object event pool. Thus, not only is the dispatching of the event occurrence asynchronously decoupled from the receipt of the event occurrence, but actual transmission of the event occurrence to the target object is asynchronously decoupled from the execution that initiated the sending of the event occurrence.

Note: The above model of the sending of event occurrences supports the general approach to the semantics of inter-object communications in fUML (see 2.3). Since each event occurrence is sent using a concurrently executing behavior in the execution model, the semantics of concurrency (as also discussed in 2.3) allows event occurrences that have been sent concurrently to be arbitrarily re-ordered in time before delivery or arbitrarily delayed in time relative to the concurrent execution of the target object event dispatch loop behaviors. This is consistent with the allowed possibility that inter-object communication may not be reliable or deterministic.

A classifier behavior for an active object may be started using a start object behavior action (see 8.10.2.40). When a behavior of an active object is so started, if no object activation yet exists for the active object, one is created. An active object with multiple types may have multiple classifier behaviors, which may be started separately, so it is possible that an object activation may already exist when a classifier behavior is started, if it is not the first one. In either case, the actual starting of the behavior is then delegated to the object activation.

To start a classifier behavior, an Execution instance is created for it (see 8.8.2.7), but this execution does not run immediately. Instead, an invocation event occurrence for the execution is added to the event pool and a classifier behavior invocation event accepter is registered to handle this event occurrence. As previously described in general for event occurrences, this decouples the receipt of the event occurrence requesting the start of a classifier behavior from the dispatching of the event occurrence, at which point the classifier behavior invocation event accepter actually starts the classifier behavior execution. In this way, the classifier behavior executes asynchronously from its invocation and within an initial run-to-completion step, so that any event occurrences received by the active object during this initial execution are saved until the object is ready to handle them. (Note that an invocation event occurrence is not sent using the asynchronous EventOccurrence behavior, as described above for signal and call event occurrences, but is placed directly into the event pool of the context object of the classifier behavior execution.)

An object activation also keeps a list of the classifier behavior invocation event accepters created to start classifier behavior executions. This maintains a link between the object activation and any ongoing executions so that, if the associated active object is destroyed, any running classifier behavior executions may be terminated.

Once a classifier behavior is running, it may register event accepters to handle the occurrences of other kinds of events received by its context object. In particular, the firing of an accept event action results in the registration of an event accepter for the events declared in the triggers of that action (see 8.10.2.2 and 8.10.2.3). Currently, a regular accept event action in fUML is limited to handling signal events, while an accept call action (a special kind of accept event action) is used to handle call events (see 8.10.2.1). When a signal or call event occurrence is received by an active object, it is placed in the event pool. When this event occurrence is dispatched, if there is a matching accept event action accepter for it, then it will be accepted by the accept event action (or accept call action, for a call event occurrence), resulting in the resumption of execution of the activity containing the action.

As mentioned above under Event Dispatching, an operation call is always a synchronous invocation from the point of view of the caller, even if it is handled asynchronously using a call event at the target. In order to achieve this, a call event occurrence is always sent by executing a special call event execution object (see 8.8.2.2), after which the calling execution thread is blocked until a reply to the call is received. Blocking is specified in the suspend operation of the call event execution using a loop that repeatedly checks for the callSuspended flag to be reset by the concurrent thread responding to the call. The body of the loop consists of a call to a special wait operation, which does nothing, but, during which, a conforming execution tool must allow other concurrent threads to run. That is, an execution trace that, after a certain point, consists entirely of a caller executing one or more suspend loops for all time is not allowed, unless no other execution trace is possible (i.e., no other non-blocked concurrent threads are available to execute).

Note: The special rule above concerning waiting is necessary because fUML allows great flexibility in whether a conforming execution tool actually implements a concurrent thread as a parallel execution or not (see 2.3). Without this rule, it would be allowable for a conforming execution tool to consume all processing resources executing one or more suspend loops and never allow the execution of the dispatch loops necessary to handle the call event occurrences that have been sent. Even with the rule, however, it is not inherently guaranteed that any call event occurrence will ever be dispatched, since the dispatch loop of the target object that receive it may still be blocked forever, in a particular implementation, by some other unsuspended thread.

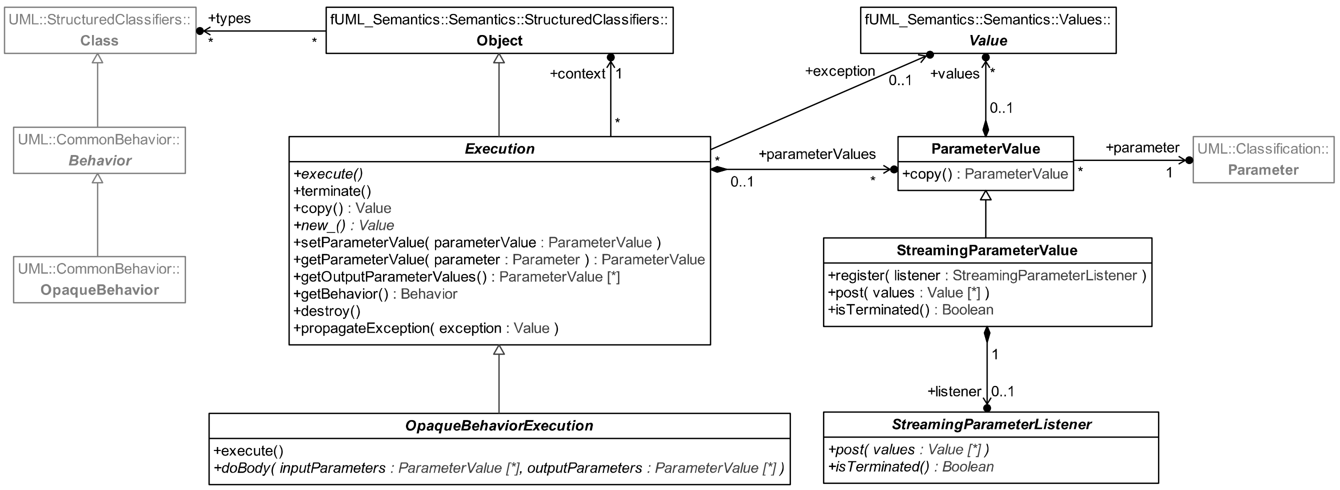

UML::StructuredClassifiers::
Class
+types
fUML_Semantics::Semantics::StructuredClassifiers::
Object
fUML_Semantics::Semantics::Values::
Value
+context
+exception
+values
UML::CommonBehavior::
Behavior
Execution
+execute()
+terminate()
+copy() : Value
+new_() : Value
+setParameterValue(parameterValue : ParameterValue)
+getParameterValue(parameter : Parameter) : ParameterValue
+getOutputParameterValues() : ParameterValue [*]
+getBehavior() : Behavior
+destroy()
+propagateException(exception : Value)
+parameterValues
ParameterValue
+copy() : ParameterValue
+parameter
UML::Classification::
Parameter
UML::CommonBehavior::
OpaqueBehavior
StreamingParameterValue
+register(listener : StreamingParameterListener)
+post(values : Value [*])
+isTerminated() : Boolean
+listener
OpaqueBehaviorExecution
+execute()
+doBody(inputParameters : ParameterValue [*], outputParameters : ParameterValue [*])
StreamingParameterListener
+post(values : Value [*])
+isTerminated() : Boolean

Figure 8.15 - Executions

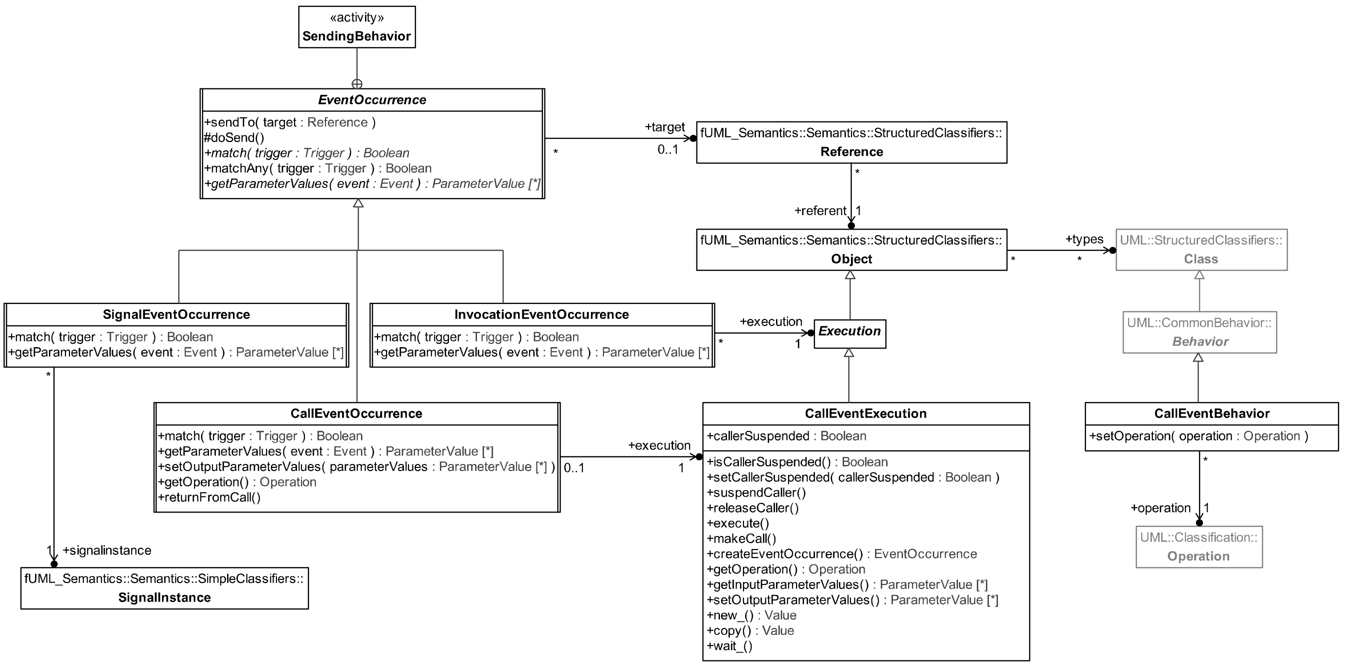

«activity»
SendingBehavior
EventOccurrence
+sendTo(target : Reference)
#doSend()
+match(trigger : Trigger) : Boolean
+matchAny(trigger : Trigger) : Boolean
+getParameterValues(event : Event) : ParameterValue [*]
+target
fUML_Semantics::Semantics::StructuredClassifiers::
Reference
+referent
fUML_Semantics::Semantics::StructuredClassifiers::
Object
+types
UML::StructuredClassifiers::
Class
SignalEventOccurrence
+match(trigger : Trigger) : Boolean
+getParameterValues(event : Event) : ParameterValue [*]
InvocationEventOccurrence
+match(trigger : Trigger) : Boolean
+getParameterValues(event : Event) : ParameterValue [*]
+execution
Execution
UML::CommonBehavior::
Behavior
CallEventOccurrence
+match(trigger : Trigger) : Boolean
+getParameterValues(event : Event) : ParameterValue [*]
+setOutputParameterValues(parameterValues : ParameterValue [*])
+getOperation() : Operation
+returnFromCall()
CallEventExecution
+callerSuspended : Boolean
+isCallerSuspended() : Boolean
+setCallerSuspended(callerSuspended : Boolean)
+suspendCaller()
+releaseCaller()
+execute()
+makeCall()
+createEventOccurrence() : EventOccurrence
+getOperation() : Operation
+getInputParameterValues() : ParameterValue [*]
+setOutputParameterValues() : ParameterValue [*]
+new_() : Value
+copy() : Value
+wait_()
CallEventBehavior
+setOperation(operation : Operation)
+operation
UML::Classification::
Operation
+signalinstance
fUML_Semantics::Semantics::SimpleClassifiers::
SignalInstance

Figure 8.16 - Event Occurrences

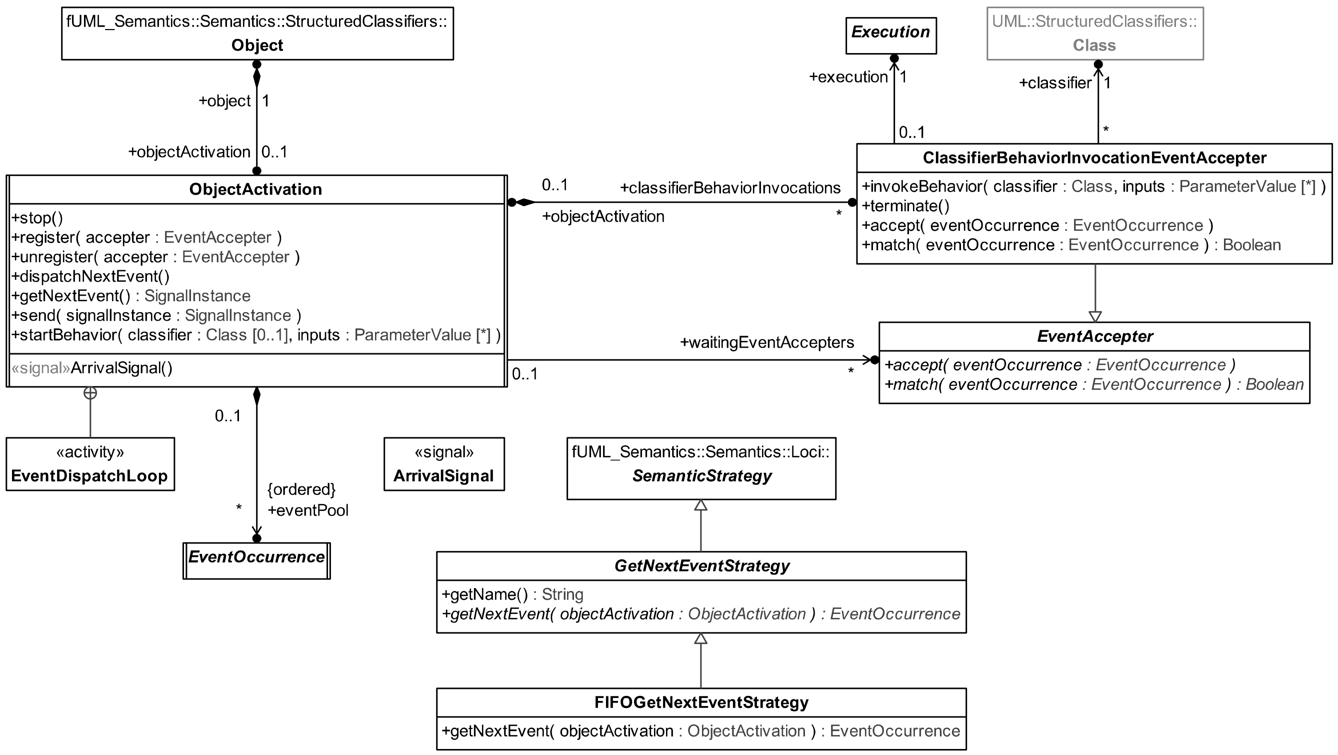

fUML_Semantics::Semantics::StructuredClassifiers::
Object
+object 1
+objectActivation 0..1
ObjectActivation
+stop()
+register(accepter : EventAccepter)
+unregister(accepter : EventAccepter)
+dispatchNextEvent()
+getNextEvent() : SignalInstance
+send(signalInstance : SignalInstance)
+startBehavior(classifier : Class [0..1], inputs : ParameterValue [*])
«signal»ArrivalSignal()
Execution
+execution 1
0..1
UML::StructuredClassifiers::
Class
+classifier 1
*
ClassifierBehaviorInvocationEventAccepter
+invokeBehavior(classifier : Class, inputs : ParameterValue [*])
+terminate()
+accept(eventOccurrence : EventOccurrence)
+match(eventOccurrence : EventOccurrence) : Boolean
0..1
+classifierBehaviorInvocations
+objectActivation
*
EventAccepter
+accept(eventOccurrence : EventOccurrence)
+match(eventOccurrence : EventOccurrence) : Boolean
+waitingEventAccepters
0..1
*
0..1
«activity»
EventDispatchLoop
{ordered}
+eventPool
*
EventOccurrence
«signal»
ArrivalSignal
fUML_Semantics::Semantics::Loci::
SemanticStrategy
GetNextEventStrategy
+getName() : String
+getNextEvent(objectActivation : ObjectActivation) : EventOccurrence
FIFOGetNextEventStrategy
+getNextEvent(objectActivation : ObjectActivation) : EventOccurrence

Figure 8.17 - Active Objects

8.8.2 Class Descriptions

8.8.2.1 CallEventBehavior

A call event behavior is a special kind of behavior used to represent the type of a call event execution. It is not directly a part of a user model, but has a signature constructed from the signature of the operation being called.

Generalizations

- Behavior

Attributes

None

Associations

- operation : Operation
  The operation whose call is to be handled via a call event.

Operations

[1] setOperation (in operation : Operation)

```
// Set the operation for this call event behavior and construct
// the behavior signature based on the operation signature.

this.operation = operation;
for(int i = 0; i < operation.ownedParameter.size(); i++){
     Parameter operationParameter = operation.ownedParameter.get(i);
     Parameter parameter = new Parameter();
     parameter.name = operationParameter.name;
     parameter.type = operationParameter.type;
     parameter.multiplicityElement.lowerValue =
               operationParameter.multiplicityElement.lowerValue;
     parameter.multiplicityElement.lower =
               operationParameter.multiplicityElement.lower;
     parameter.multiplicityElement.upperValue =
               operationParameter.multiplicityElement.upperValue;
     parameter.multiplicityElement.upper =
               operationParameter.multiplicityElement.upper;
     parameter.direction = operationParameter.direction;
     parameter.owner = this;
     parameter.namespace = this;

     this.ownedElement.addValue(parameter);
     this.ownedMember.addValue(parameter);
     this.member.addValue(parameter);
     this.ownedParameter.addValue(parameter);
}
this.isReentrant = true;
this.name = "CallEventBehavior";
if (operation.name != null) {
     this.name = this.name + "(" + operation.name + ")";
}
```

8.8.2.2 CallEventExecution

A call event execution acts as the effective method execution for an operation call that is to be handled by a call event. When executed, a call event execution sends a call event occurrence to the target object and then suspends until a reply is received.

Generalizations

- Execution

Attributes

- callerSuspended : Boolean
  Indicates whether the caller is suspended, waiting for a reply to the sent call event occurrence. (Concurrently setting this flag to "false" releases the caller.)

Associations

None

Operations

[1] copy ( ) : Value

```
// Create a new call event execution that is a copy of this execution, with the
// caller initially not suspended.

CallEventExecution copy = (CallEventExecution)super.copy();
copy.callerSuspended = false;
return copy;
```

[2] createEventOccurrence ( ) : EventOccurrence

```
// Create a call event occurrence associated with this call event execution.
// (This operation may be overridden in subclasses to alter how the event
// occurrence is create, e.g., if it is necessary to wrap it.)

CallEventOccurrence eventOccurrence = new CallEventOccurrence();
eventOccurrence.execution = this;
return eventOccurrence;
```

[3] execute( )

```
// Make the call on the target object (which is the context of this execution)
// and suspend the caller until the call is completed.

// Note: The callerSuspended flag needs to be set before the call is made,
// in case the call is immediately handled and returned, even before the
// suspend loop is started.
this.setCallerSuspended(true);

this.makeCall();
this.suspendCaller();
```

[4] getInputParameterValues ( ) : ParameterValue [0..*]

```
// Return input parameter values for this execution.

ParameterValueList parameterValues = new ParameterValueList();
for(int i=0; i < this.parameterValues.size(); i++){
     ParameterValue parameterValue = this.parameterValues.get(i);
     if(parameterValue.parameter.direction == ParameterDirectionKind.in
              | parameterValue.parameter.direction == ParameterDirectionKind.inout){
          parameterValues.addValue(parameterValue);
     }
}
return parameterValues;
```

[5] getOperation ( ) : Operation

```
// Return the operation being called by this call event execution.

return ((CallEventBehavior)this.getBehavior()).operation;
```

[6] isCallerSuspended ( ) : Boolean

```
// Check if the caller is still suspended.
// This is done in isolation from possible concurrent updates to this flag.

_beginIsolation();
boolean isSuspended = this.callerSuspended;
_endIsolation();

return isSuspended;
```

[7] makeCall ( )

```
// Make the call on the target object (which is the context of this execution)
// by sending a call event occurrence. (Note that the call will never be
// completed if the target is not an active object, since then the object
// would then have no event pool in which the event occurrence could be placed.)

Reference reference = new Reference();
reference.referent = this.context;
this.createEventOccurrence().sendTo(reference);
```

[8] new_ ( ) : Value

```
// Create a new call event execution.

return new CallEventExecution();
```

[9] releaseCaller ( )

```
// Release the caller, if suspended.
```

```
this.setCallerSuspended(false);
```

[10] setCallerSuspended (in callerSuspended : Boolean)

```
// Set the caller suspended flag to the given value.
// This is done in isolation from possible concurrent queries to this flag.

_beginIsolation();
this.callerSuspended = callerSuspended;
_endIsolation();
```

[11] setOutputParameterValues (parameterValues : ParameterValue [0..*])

```
// Set the output parameter values for this execution.

ParameterList parameters = this.getBehavior().ownedParameter;
int i = 1;
int j = 1;
while (i <= parameters.size()) {
     Parameter parameter = parameters.get(i-1);
     if (parameter.direction == ParameterDirectionKind.inout |
               parameter.direction == ParameterDirectionKind.out |
                    parameter.direction == ParameterDirectionKind.return_ ) {
          ParameterValue parameterValue = parameterValues.get(j-1);
          parameterValue.parameter = parameter;
          this.setParameterValue(parameterValue);
          j = j + 1;
     }
     i = i + 1;
}
```

[12] suspendCaller ( )

```
// Suspend the caller until the caller is released.

while(this.isCallerSuspended()) {
     this.wait_();
}
```

[11] wait_ ( )

```
// Wait for an indeterminate amount of time to allow other concurrent
// executions to proceed.
// [There is no further formal specification for this operation.]
```

8.8.2.3 CallEventOccurrence

A call event occurrence represents the occurrence of a call event due to a call to a specific operation.

Generalizations

- EventOccurrence

Attributes

None

Associations

- execution : CallEventExecution
  The call event execution that created this call event occurrence.

Operations

[1] getOperation ( ) : Operation

```
// Get the operation being called by this call event occurrence.

return this.execution.getOperation();
```

[2] getParameterValues (in event : Event) : ParameterValue [0..*]

```
// Return the input parameter values from the call event execution for
// this call event occurrence, which correspond to the values of the
// operation input parameters for the call.

return this.execution.getInputParameterValues();
```

[3] match (trigger : Trigger) : Boolean

```
// Match a trigger if it references a call event whose operation is the
// operation of this call event occurrence.

boolean matches = false;
if (trigger.event instanceof CallEvent) {
     CallEvent callEvent = (CallEvent)trigger.event;
     matches = callEvent.operation == this.getOperation();
}
return matches;
```

[4] releaseCaller ( )

```
// Release the caller on return from the call.

this.execution.releaseCaller();
```

[5] setOutputParameterValues (parameterValues : ParameterValue [0..*])

```
// Set the output parameter values of the call event execution for
// this call event occurrence, which correspond to the values of the
// operation output parameters for the call.

this.execution.setOutputParameterValues(parameterValues);
```

8.8.2.4 ClassifierBehaviorInvocationEventAccepter

A classifier behavior accepts an invocation event occurrence for the invocation of the execution of a classifier behavior from a specific active class.

Generalizations

- EventAccepter

Attributes

None

Associations

- classifier : Class
  The classifier whose behavior is being executed. (This must be an active class.)
- execution : Execution
  The execution of the associated classifier behavior for a certain object.
- objectActivation : ObjectActivation [0..1]
  The object activation that owns this classifier behavior execution.

Operations

[1] accept (in eventOccurrence : EventOccurrence)

```
// Accept an invocation event occurrence. Execute the execution of this
// classifier behavior invocation event accepter.

if (eventOccurrence instanceof InvocationEventOccurrence) {
     this.execution.execute();
}
```

[2] invokeBehavior (in classifier : Class, in inputs : ParameterValue [0..*])

```
// Set the classifier for this classifier behavior execution to the given class.
// If the given class is a behavior, set the execution to be the object of the object
activation of the classifier behavior execution.
// Otherwise the class must be an active class, so get an execution object for the classifier
behavior for the class.
// Set the input parameters for the execution to the given values.
// Then register this event accepter with the object activation.

this.classifier = classifier;
Object_ object = this.objectActivation.object;

if (classifier instanceof Behavior) {
     this.execution = (Execution)object;
} else {
      this.execution = object.locus.factory.createExecution(classifier.classifierBehavior,
object);
}

if (inputs != null) {
     for (int i = 0; i < inputs.size(); i++) {
          ParameterValue input = inputs.getValue(i);
          this.execution.setParameterValue(input);
     }
}

this.objectActivation.register(this);
```

[3] match (in eventOccurrence : EventOccurrence) : Boolean

```
// Return true if the given event occurrence is an invocation event
// occurrence for the execution of this classifier behavior invocation
// event accepter.

boolean matches = false;
if (eventOccurrence instanceof InvocationEventOccurrence) {
     matches = ((InvocationEventOccurrence)eventOccurrence).execution == this.execution;
}
return matches;
```

[4] terminate ( )

```
// Terminate the associated execution.
// If the execution is not itself the object of the object activation, then destroy it.
```

```
this.execution.terminate();

if (this.execution != this.objectActivation.object) {
     this.execution.destroy();
}
```

8.8.2.5 EventAccepter

An event accepter handles signal reception events.

This is an abstract class intended to provide a common interface for different kinds of event accepters.

Generalizations

None

Attributes

None

Associations

- None

Operations

[1] accept (in eventOccurrence : EventOccurrence)

```
Accept a signal occurrence for the given signal instance.
```

[2] match (in eventOccurrence : EventOccurrence) : Boolean

```
Determine if the given signal instance matches a trigger of this event accepter.
```

8.8.2.6 EventOccurrence

An event occurrence represents a single occurrence of a specific kind of event.

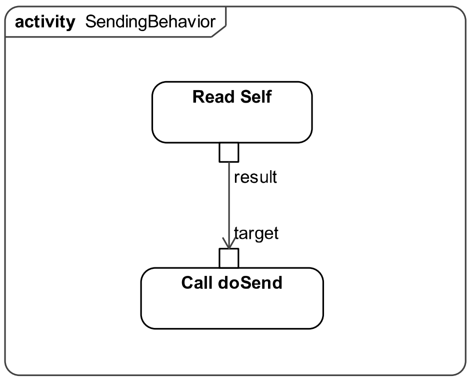


Figure 8.18 - Sending Behavior

Generalizations

None

Attributes

None

Associations

- target : Reference [0..1]
  A reference to the target object to which this event occurrence is being sent.

Operations

[1] doSend ( )

```
// Send this event occurrence to the target reference.

this.target.send(this);
```

[2] getParameterValues (in event : Event) : ParameterValue[0..*]

```
Return the values of parametric data associated with this event occurrence relevant to the
given event.
```

[3] match (in trigger : Trigger) : Boolean

```
Return true if this event occurrence matches the given trigger. Each concrete specialization of
EventOccurrence must provide a behavior for this operation.
```

[4] matchAny (in triggers : Trigger [0..*]) : Boolean

```
// Check that at least one of the given triggers is matched by this
// event occurrence.

boolean matches = false;
int i = 1;
while(!matches & i <= triggers.size()){
     if(this.match(triggers.get(i-1))){
          matches = true;
     }
     i = i + 1;
}
return matches;
```

[5] sendTo (in target : Reference)

```
// Set the target reference and start the SendingBehavior, which
// will send this event occurrence to the target.

this.target = target;
_startObjectBehavior();
```

8.8.2.7 Execution

An execution is used to execute a specific behavior. Since a behavior is a kind of class, an execution is an object with the behavior as its type.

Generalizations

- Object

Attributes

None

Associations

- context : Object
  The object that provides the context for this execution.
  The type of the context of the execution must be the context of the type (behavior) of the execution.
- exception : Value [0..1]
  The value raised as an exception by this execution, if any.
- parameterValues : ParameterValue [0..*]
  The parameterValues for this execution. All parameterValues must have a parameter that is a parameter of the type of this execution.
  The values of all input (in and in-out) parameters must be set before the execution is executed.

Operations

[1] copy ( ) : Value

```
// Create a new execution that has the same behavior and parameterValues as this execution.

Execution newValue = (Execution)(super.copy());

newValue.context = this.context;

ParameterValueList parameterValues = this.parameterValues;
for (int i = 0; i < parameterValues.size(); i++) {
     ParameterValue parameterValue = parameterValues.getValue(i);
     newValue.parameterValues.addValue(parameterValue.copy());
}

return newValue;
```

[2] destroy ( )

```
// Terminate the execution before destroying it.

this.terminate();
super.destroy();
```

[3] execute ( )

```
Execute the behavior given by the type of this execution.
The parameterValues for any input (in or in-out) parameters of the behavior should be set
before the execution.
The parameterValues for any output (in-out, out or return) parameters of the behavior will be
set by the execution.
```

[4] getBehavior ( ) : Behavior

```
// Get the behavior that is the type of this execution.

return  (Behavior)(this.getTypes().getValue(0));
```

[5] getOutputParameterValues ( ) : ParameterValue [0..*]

```
// Return the parameter values for output (in-out, out and return) parameters.

ParameterValueList outputs = new ParameterValueList();
ParameterValueList parameterValues = this.parameterValues;
for (int i = 0; i < parameterValues.size(); i++) {
     ParameterValue parameterValue = parameterValues.getValue(i);
     Parameter parameter = parameterValue.parameter;
     if ((parameter.direction == ParameterDirectionKind.inout) |
          (parameter.direction == ParameterDirectionKind.out) |
          (parameter.direction == ParameterDirectionKind.return_)) {
          outputs.addValue(parameterValue);
     }
}

return outputs;
```

[6] getParameterValue (in parameter : Parameter) : ParameterValue

```
// Get the parameter value of this execution corresponding to the given parameter (if any).

ParameterValue parameterValue = null;
int i = 1;
while (parameterValue == null & i <= this.parameterValues.size()) {
     if (this.parameterValues.getValue(i-1).parameter == parameter) {
          parameterValue = this.parameterValues.getValue(i-1);
     }
     i = i + 1;
}

return parameterValue;
```

[7] new_ ( ) : Value

```
Create a new execution with no behavior or parameterValues.
```

[8] propagateException (in exception : Value)

```
// Set the propagated exception for this execution to the given exception,
// then terminate the execution.

this.exception = exception;
this.terminate();
```

[9] setParameterValue (in parameterValue : ParameterValue)

```
// Set the given parameter value for this execution.
// If a parameter value already existed for the parameter of the given parameter value, then
replace its value.

ParameterValue existingParameterValue = this.getParameterValue(parameterValue.parameter);

if (existingParameterValue == null) {
     this.parameterValues.addValue(parameterValue);
}
else {
     existingParameterValue.values = parameterValue.values;
}
```

[10] terminate ( )

```
// Terminate an ongoing execution. By default, do nothing.

return;
```

8.8.2.8 FIFOGetNextEventStrategy

A FIFO get next event strategy gets events in first-in first-out order.

Generalizations

- GetNextEventStrategy

Attributes

None

Associations

None

Operations

[1] getNextEvent (in objectActivation : ObjectActivation) : EventOccurrence

```
// Get the first event from the given event pool. The event is removed from the pool.

EventOccurrence eventOccurrence = objectActivation.eventPool.getValue(0);
objectActivation.eventPool.removeValue(0);
return eventOccurrence;
```

8.8.2.9 GetNextEventStrategy

A get next event strategy is a semantic strategy that determines the order in which signal instances are retrieved from the event pool of an object activation.

Generalizations

- SemanticStrategy

Attributes

None

Associations

None

Operations

[1] getName ( ) : String

```
// Get next event strategies are always named "getNextEvent".

return "getNextEvent";
```

[2] getNextEvent (in objectActivation : ObjectActivation) : EventOccurrence

```
Get the next event from the event pool of the given object activation. The event is removed
from the pool.
```

8.8.2.10 InvocationEventOccurrence

An invocation event occurrence represents a signal occurrence of the event of the asynchronous invocation of a specific behavior execution.

Generalizations

- EventOccurrence

Attributes

None

Associations

- execution : Execution
  The execution being asynchronously invoked.

Operations

[1] getParameterValues ( ) : ParameterValue[0..*]

```
// An invocation event occurrence does not have any associated data.

return new ParameterValueList();
```

[2] match (in trigger : Trigger) : Boolean

```
// An invocation event occurrence does not match any triggers.

return false;
```

8.8.2.11 ObjectActivation

An object activation handles the active behavior of an active object.

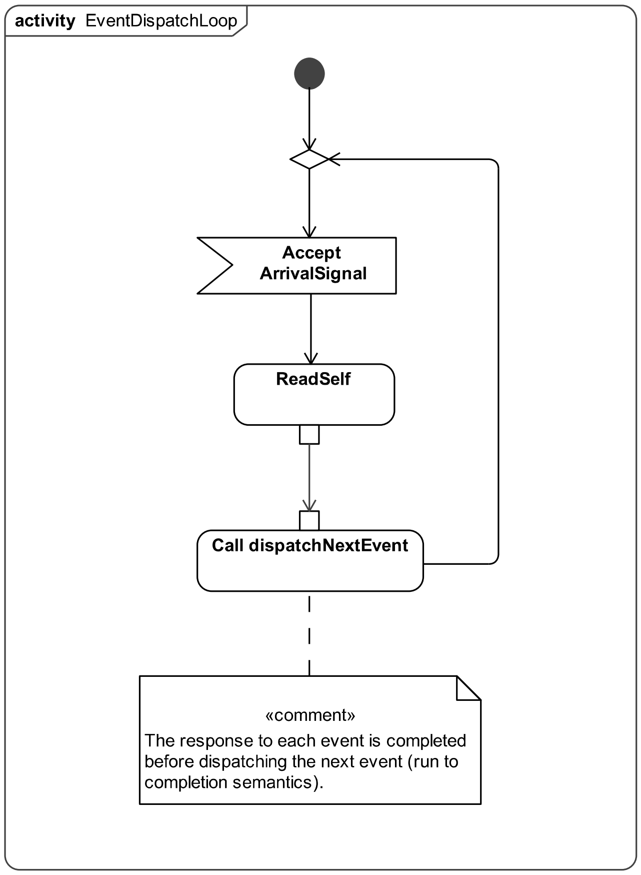

activity EventDispatchLoop
Accept ArrivalSignal
ReadSelf
Call dispatchNextEvent
«comment»
The response to each event is completed before dispatching the next event (run to completion semantics).

Figure 8.19 - Classifier Behavior for ObjectActivation

Generalizations

None

Attributes

None

Associations

- classifierBehaviorInvocations : ClassifierBehaviorInvocationEventAccepter [0..*]
  The invocations of the executing classifier behaviors for this object activation.
- eventPool : EventOccurrence [0..*]
  The pool of event occurrences received by the object of this object activation, pending dispatching.
- object : Object
  The object whose active behavior is being handled by this active object.
- waitingEventAccepters : EventAccepter [0..*]
  The set of event accepters waiting for event occurrences to be dispatched from the event pool.

Operations

[1] dispatchNextEvent ( )

```
// Get the next event occurrence out of the event pool.
// If there are one or more waiting event accepters with triggers that
// match the event occurrence, then dispatch it to exactly one of those
// waiting accepters.

if (this.eventPool.size() > 0) {
     EventOccurrence eventOccurrence = this.getNextEvent();

     intList matchingEventAccepterIndexes = new intList();
     EventAccepterList waitingEventAccepters = this.waitingEventAccepters;
     for (int i = 0; i < waitingEventAccepters.size(); i++) {
          EventAccepter eventAccepter = waitingEventAccepters.getValue(i);
          if (eventAccepter.match(eventOccurrence)) {
              matchingEventAccepterIndexes.addValue(i);
          }
     }

     if (matchingEventAccepterIndexes.size() > 0) {
          // *** Choose one matching event accepter non-deterministically. ***
          int j =
((ChoiceStrategy)this.object.locus.factory.getStrategy("choice")).choose(matchingEventAccepte
rIndexes.size());
          int k = matchingEventAccepterIndexes.getValue(j - 1);
          EventAccepter selectedEventAccepter = this.waitingEventAccepters.getValue(k);
          this.waitingEventAccepters.removeValue(k);
          selectedEventAccepter.accept(eventOccurrence);
     }
}
```

[2] getNextEvent ( ) : EventOccurrence

```
// Get the next event from the event pool, using a get next event strategy.

return
((GetNextEventStrategy)this.object.locus.factory.getStrategy("getNextEvent")).getNextEvent(th
is);
```

[3] register (in accepter : EventAccepter)

```
// Register the given event accepter to wait for a dispatched signal event.
```

```
this.waitingEventAccepters.addValue(accepter);
```

[4] send (in eventOccurrence : eventOccurrence)

```
// Add the event occurrence to the vent pool and signal that a
// new event occurrence has arrived.

this.eventPool.addValue(eventOccurrence);
_send(new  ArrivalSignal());
```

[5] startBehavior (in classifier : Class [0..1], in inputs : ParameterValue [0..*])

```
// Start the event dispatch loop for this object activation (if it has not already been
started).
// If a classifier is given that is a type of the object of this object activation and there
is not already a classifier behavior invocation for it,
//      then create a classifier behavior invocation for it.
// Otherwise, create a classifier behavior invocation for each of the types of the object of
this object activation which has a classifier behavior or which is a behavior itself
//      and for which there is not currently a classifier behavior invocation.

// Start EventDispatchLoop
_startObjectBehavior();

if (classifier == null) {
     // *** Start all classifier behaviors concurrently. ***
     Class_List types = this.object.types;
     for (Iterator i = types.iterator(); i.hasNext();) {
         Class_ type = (Class_)i.next();
         if (type instanceof Behavior | type.classifierBehavior != null) {
              this.startBehavior(type, new ParameterValueList());
         }
     }
}
else {
     _beginIsolation();
     boolean notYetStarted = true;
     int i = 1;
     while (notYetStarted & i <= this.classifierBehaviorInvocations.size()) {
          notYetStarted = (this.classifierBehaviorInvocations.getValue(i-1).classifier !=
classifier);
         i = i + 1;
     }
```

```
    if (notYetStarted) {
      ClassifierBehaviorInvocationEventAccepter newInvocation =
          new ClassifierBehaviorInvocationEventAccepter();
      newInvocation.objectActivation = this;
      this.classifierBehaviorInvocations.addValue(newInvocation);
      newInvocation.invokeBehavior(classifier, inputs);
      InvocationEventOccurrence eventOccurrence = new InvocationEventOccurrence();
      eventOccurrence.execution = newInvocation.execution;
      this.eventPool.addValue(eventOccurrence);
      _send(new ArrivalSignal());
    }
    _endIsolation();
}
```

[6] stop ( )

```
// Stop this object activation by terminating all classifier behavior executions.

ClassifierBehaviorInvocationEventAccepterList  classifierBehaviorInvocations  =
this.classifierBehaviorExecutions;
for (int i = 0; i < classifierBehaviorExecutions.size(); i++) {
     ClassifierBehaviorInvocationEventAccepter classifierBehaviorInvocation =
classifierBehaviorExecutions.getValue(i);
     classifierBehaviorExecution.terminate();
}
```

[7] unregister (in accepter : EventAccepter)

```
// Remove the given event accepter for the list of waiting event accepters.

boolean notFound = true;
int i = 1;
while (notFound & i <= this.waitingEventAccepters.size()) {
     if (this.waitingEventAccepters.getValue(i-1) == accepter) {
         this.waitingEventAccepters.remove(i-1);
         notFound = false;
     }
     i = i + 1;
}
```

8.8.2.12 OpaqueBehaviorExecution

An opaque execution is an execution for an opaque behavior.

Opaque behaviors are used to define primitive behaviors.

The actual definition of the primitive behavior should be given in a concrete subclass of OpaqueBehaviorExecution.

Generalizations

- Execution

Attributes

None

Associations

None

Operations

[1] doBody (in inputParameters : ParameterValue [0..*], in outputParameters : ParameterValue [0..*])

```
The actual definition of the behavior of an Opaque Behavior should be given in a concrete
subclass that defines this operation.
The values of the inputParameters are set when the operation is called.
The values of the outputParmeters should be set during the execution of the operation.
```

[2] execute ( )

```
// Execute the body of the opaque behavior.

ParameterList parameters = this.getBehavior().ownedParameter;

ParameterValueList inputs = new ParameterValueList();
ParameterValueList outputs = new ParameterValueList();

for (int i = 0; i < parameters.size(); i++) {
     Parameter parameter = parameters.getValue(i);

     if ((parameter.direction == ParameterDirectionKind.in) |
          (parameter.direction == ParameterDirectionKind.inout)) {
          inputs.addValue(this.getParameterValue(parameter));
     }

     if ((parameter.direction == ParameterDirectionKind.inout) |
          (parameter.direction == ParameterDirectionKind.out) |
          (parameter.direction == ParameterDirectionKind.return_)) {
```

```
        ParameterValue parameterValue = new ParameterValue();
        parameterValue.parameter = parameter;
        this.setParameterValue(parameterValue);
        outputs.addValue(parameterValue);
    }
}

this.doBody(inputs, outputs);
```

8.8.2.13 ParameterValue

A parameter value gives the value(s) for a specific parameter.

Generalizations

None

Attributes

None

Associations

- parameter : Parameter [0..1]
  The parameter for which values are being provided. (This may be empty in the case of an internally generated "effective" parameter value, e.g., to represent data extracted from a SignalEventOccurrence.)
- values : Value [0..*]
  The values for the parameter. Zero or more values are possible, as constrained by the multiplicity of the parameter.

Operations

[1] copy ( ) : ParameterValue

```
// Create a new parameter value for the same parameter as this parameter value, but with
copies of the values of this parameter value.

ParameterValue newValue = new ParameterValue();

newValue.parameter = this.parameter;

ValueList values = this.values;
for (int i = 0; i < values.size(); i++) {
    Value value = values.getValue(i);
    newValue.values.addValue(value.copy());
}

return newValue;
```

8.8.2.14 SignalEventOccurrence

A signal event occurrence represents the occurrence of a signal event due to the receipt of a specific signal instance.

Generalizations

- EventOccurrence

Attributes

None

Associations

- signalInstance : SignalInstance
  The signal instance whose receipt caused this signal event occurrence.

Operations

[1] getParameterValues (in event : Event) : ParameterValue[0..*]

```
// Return parameter values for the features of the signal instance, in order,
// corresponding to the attributes of the declared signal of the given event.
// These are intended to be treated as if they are the values of effective
// parameters of direction "in".
// (Note that the given event must be a signal event, and the signal instance
// of this signal event occurrence must be a direct or indirect instance of
// the event signal.)

ParameterValueList parameterValues = new ParameterValueList();
if (event instanceof SignalEvent) {
    StructuralFeatureList memberFeatures =
            this.signalInstance.getMemberFeatures(((SignalEvent)event).signal);
    for(int i = 0; i < memberFeatures.size(); i++){
        StructuralFeature feature = memberFeatures.getValue(i);
        ParameterValue parameterValue = new ParameterValue();
        parameterValue.values = this.signalInstance.getFeatureValue(feature).values;
        parameterValues.add(parameterValue);
    }
}
return parameterValues;
```

[2] match (in trigger : Trigger) : Boolean

```
// Match a trigger if it references a signal event whose signal is the type of the
// signal instance or one of its supertypes.
```

```
boolean matches = false;
if(trigger.event instanceof SignalEvent){
     SignalEvent event = (SignalEvent) trigger.event;
     matches = this.signalInstance.isInstanceOf(event.signal);
}
return matches;
```

8.8.2.15 StreamingParameterListener

A streaming parameter listener handles the posting of values from a streaming parameter value to some target.

This is an abstract class intended to provide a common listener interface for different kinds of targets.

Generalizations
None

Attributes
None

Associations
None

Operations

[1] isTerminated ( ) : Boolean

```
Check whether the target of this listener has terminated.
```

[2] post (values : Value [0..*])

```
Post the given values to the target of this listener.
```

8.8.2.16 StreamingParameterValue

Generalizations

- ParameterValue

Attributes

None

Associations

- listener : StreamingParameterListener [0..1]
  The listener for values from this streaming parameter value. A streaming parameter value can have at most one such listener.

Operations

[1] isTerminated ( ) : Boolean

```
// Check if this streaming parameter value either has no listener,
// or it has a listener that has terminated.
```

```
boolean isTerminated = true;
if (this.listener != null) {
    isTerminated = this.listener.isTerminated();
}
return isTerminated;
```

[2] post (values : Value [0.*])

```
// Post the given values to the listener, if there is at
// least one value.

this.values = values;

if (this.listener != null & values.size() > 0) {
    listener.post(values);
}
```

[3] register (listener : StreamingParameterListener)

```
// Register a listener for this streaming parameter value.

this.listener = listener;
```

8.9 Activities

8.9.1 Overview

Activities are the only concrete sort of user behavior model included in fUML. (Opaque behaviors are also included in fUML, but only for specifying primitive behaviors.) Subclause 7.10 gives the abstract syntax for activities. The elements of this syntax are that activities are composed of activity nodes with control flow and object flow activity edges connecting the nodes. The present subclause describes the basic semantics of activity execution in terms of activations of the activity nodes in the activity. The semantics for actions, which are a kind of activity node, are given in 8.10.

Activity Node Activation

As shown in Figure 8.23, the activity execution model is an extension of the general behavior execution model from 8.8. In addition to activity executions themselves, the model includes activity node activations that specify the behavior of activity nodes during a specific activity execution. These node activations are then interconnected by activity edge instances corresponding to the activity edges in the activity.

Activity node activations are semantic visitor classes, like evaluations and executions (see 8.3.1 for a discussion of semantic visitor classes in general). There is an activation visitor class corresponding to each concrete subclass of ActivityNode. The name of the visitor class is the same as the name of the corresponding abstract syntax metaclass with the word "Activation" appended. For example, the activation visitor class for the abstract syntax metaclass JoinNode is called JoinNodeActivation. Note that actions are activity nodes, so that the semantics of actions are specified using activation visitor classes (see 8.10).

Activity node activations are always created within an activity node activation group. This concept is introduced in the execution model to handle nested groups within an activity. The activity itself is considered to implicitly be the top-level group.

Token and Offer Flow

Note that, consistent with the overall use of the Visitor Pattern (see 8.3.1), the activity execution model intentionally has a largely parallel structure to the abstract syntax model from 7.10. However, there are concepts introduced in the semantic model for which there is no explicit syntax in UML. In this case, the most important such concepts are those of token and offer. Consider the simple activity model shown in Figure 8.20. Figure 8.21 shows the abstract syntax representation of this model, which may then be given the semantic interpretation shown in Figure 8.22.

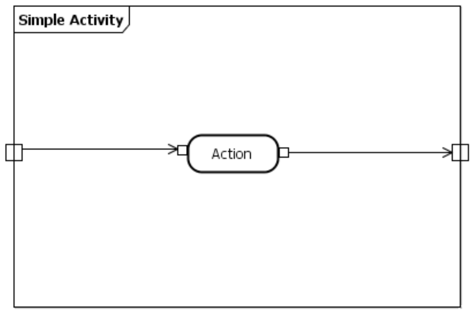


Figure 8.20 - A Simple Activity Model

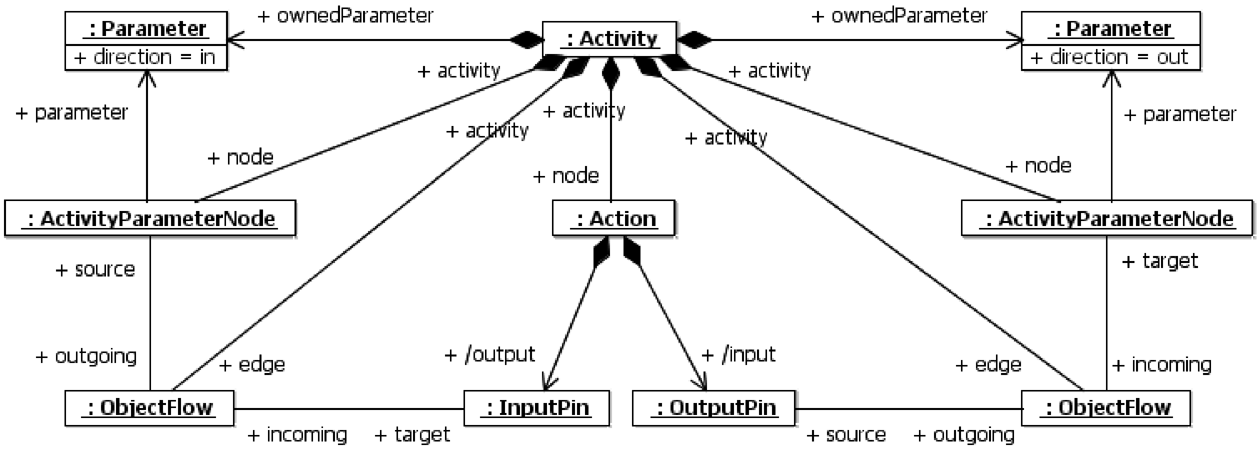


Figure 8.21 - Abstract Syntax Representation of a Simple Activity Model

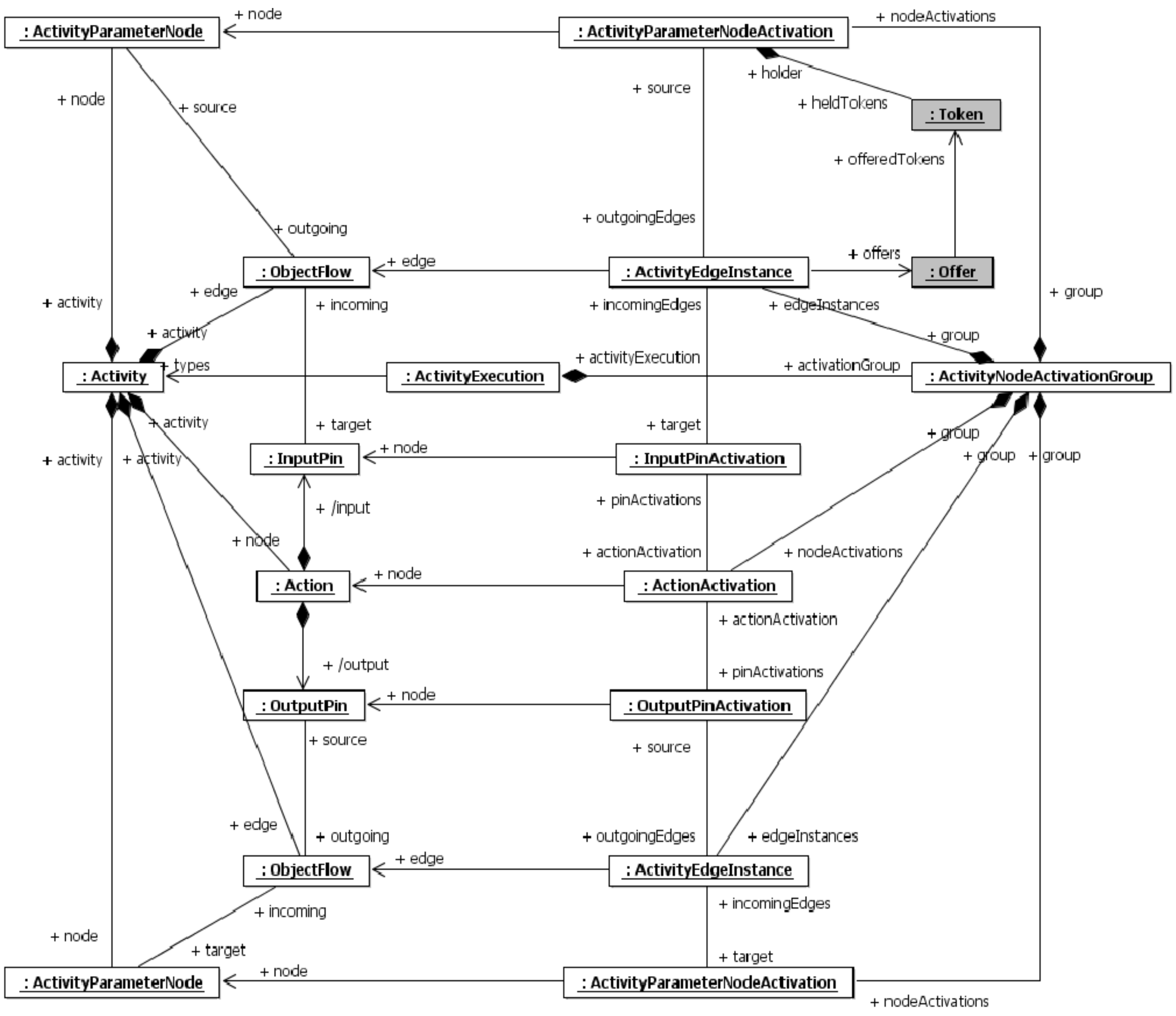


Figure 8.22 - Semantic Interpretation of a Simple Activity Model at the Start of Execution

So far, the interpretation shown in Figure 8.22 provides essentially just the structural semantics of activities, in which an activity execution is interpreted as an instance of the activity considered as a classifier. To truly capture the behavior semantics, the interpretation needs to further define how the execution of the activity proceeds over time. The UML 2 Specification defines the behavior of an activity in terms of tokens that may be held by nodes and offers made between nodes for the movement of these tokens.

The execute operation on an activity execution object places tokens on the (non-streaming) input activity parameter nodes of the activity. Figure 8.22 shows an early stage in the execution of the activity from Figure 8.20, in which the input activity parameter node holds an object token corresponding to the input parameter value for the activity execution and this node is offering the token to the input pin of the action. The behavioral semantic rules of UML activity execution then determine if and when the action will accept the offered token to its input pin.

Presuming that the input pin has multiplicity of 1 and a token for a single value has been offered, the action will accept the offer, receive the offered token on its input pin and fire its own behavior. A token with the result value from this behavior will then be placed on the output pin of the action and subsequently offered to the output parameter node. Figure 8.23 shows the semantic interpretation of this successor to the earlier stage of execution shown in Figure 8.22. The execution of this activity then concludes with the output activity parameter node accepting the offered token. At the end of the execution of an activity, the execute operation then places the values in tokens held by any (non-streaming) output activity parameter nodes onto the corresponding output parameters of the activity.

Note: In the UML abstract syntax, pins are multiplicity elements with optional ordering and so are parameters. However, while activity parameter nodes may be typed, they are not multiplicity elements and they cannot be specifically identified as ordered. Nevertheless, the fUML semantics interprets an output activity parameter node as effectively having the ordering specified for its associated parameter. Thus, when multiple tokens flow from an ordered output pin to an output activity parameter node, this ordering is preserved when the values on the tokens are ultimately placed on the corresponding output parameter.

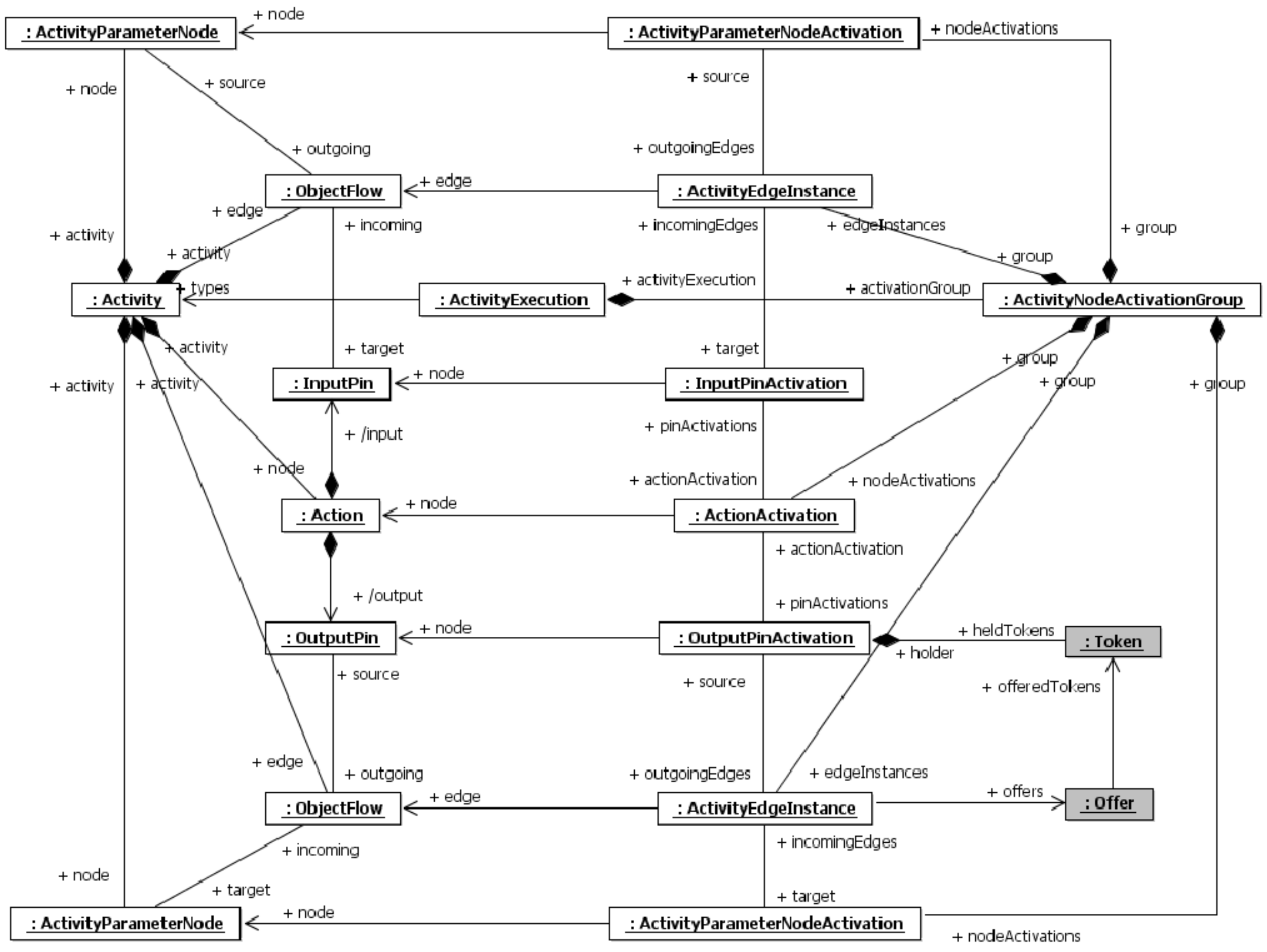


Figure 8.23 - Semantic Interpretation of a Simple Activity Model Just Prior to Completion of Execution

Threading Model

The execution semantics for activities in UML places no restriction on the concurrent activation of activity nodes within an activity, other than that imposed by the semantics of token and offer flow across the activity edges connecting the nodes. The execution model captures this concurrent execution semantics through an implicit concept of threading.

When an activity node activation produces tokens and is ready to offer them to downstream activations, it calls the sendOffer operation on outgoing activity edge instances. The edge instance sendOffer operation, in turn, signals to the target activity node activation that an offer is available by calling the receiveOffer operation. The target activity node activation then checks if its execution prerequisites are satisfied (encoded in the method of the isReady operation for each kind of activity node activation) and, if so, it accepts the pending offers made to it using its takeOfferedTokens operation and then calls its fire operation.

Note that, in the execution model, the self-calls to the isReady operation and, if the activation is ready, to the takeOfferedTokens operation happen within a single isolated region-that is, a structured activity node with mustIsolate = true. This ensures that, if the takeOfferedTokens operation is invoked, then any offers checked by the isReady operation cannot be accepted by any other activity node activation before the takeOfferedTokens operation completes. The invocation of the fire operation, however, does not occur within this isolated region, in order to not block continued concurrency with other activity node activations. (See 8.10.1 for a discussion of the semantics of structured activity nodes with mustIsolate = true.)

The method of the fire operation for an activity node activation captures the execution behavior of the corresponding activity node, which may then cause new offers to be sent further downstream. While there is no explicit class for it in the Execution Model, an extended chain of sendOffer-receiveOffer-fire-sendOffer calls can be considered to be a single thread of execution through an activity.

When an activity begins execution, a control token is implicitly placed on each enabled node. Enabled nodes include initial nodes, input activity parameter nodes, and actions with no incoming control nodes or input pins. If such an enabled node is immediately ready to fire, then it begins an execution thread within the activity execution. If there is more than one enabled node that fires, then each one begins a concurrent thread within the activity execution.

Note: The UML 2 Specification (subclause 15.2.3.6) states that "When an Activity is first invoked…A single control token is placed on each enabled node and they begin executing concurrently. Such nodes include ExecutableNodes…with no incoming ControlFlows and no mandatory input data…." Actions are kinds of executable nodes, which has "mandatory input data" if it has input pins, at least one of which has a multiplicity lower bound greater than zero. On the other hand, if the action has input pins, but they all have multiplicity lower bounds of zero, then placing a control token on the action will cause it to fire immediately. However, this is likely not to be the expected behavior, since, having input pins, the presumption is that the modeler expected the action to have at least some input. Therefore, fUML requires that an action with input pins have an offer on at least one of the pins before it fires, even if all the input pins have zero multiplicity lower bound.

It is also possible for a thread to split. This occurs whenever the same offer is made to multiple outgoing edges, such as when there are multiple edges leaving an output pin, fork node or action. Again, each outgoing thread executes concurrently-which is modeled by requiring that the sendOffer calls on outgoing edges are all made concurrently.

Note: This model of execution concurrency does not require the implementation of actual parallelism in a conforming execution tool. It simply means that such parallelism is allowed and that the execution semantics provide no further restriction on the serialization of execution across concurrent threads.

A thread ends when a target activity node activation does not accept an offer passed to it along the thread. In this case, the receiveOffer operation on the target node activation returns without calling the fire operation, and the chain of calls making up the thread terminates. For example, the input pin of an action cannot accept an offer unless its action as a whole is ready to execute (see 8.10.1). Therefore, if an action has several input pins with non-zero multiplicity lower bound, then offers need to be delivered to every input pin before the action can execute. Thus, all the threads delivering these offers, except the last one, will terminate at the action input pin activations. Only the thread delivering the final offer (assuming all the other offered tokens are still available) will result in the action firing, with the action execution continuing on that thread.

If an activity has no streaming parameters, an execution of the activity terminates when all threads within it have ended or with the raising of an exception. Such termination may happen naturally when, for example, all tokens are consumed by nodes that do not produce any new offers, or it may be forced by an activity final node. When an activity final node fires, it causes its enclosing activity execution to call the terminate operation on all activity node activations within it. Once a node activation is terminated, it will no longer accept any offers and, as a result, all executing threads will eventually end, resulting in the termination of the activity execution. When an exception is raised, the activity execution terminates in the same way as in the case of the firing of an activity final node, but it also records the exception value that caused the termination.

If an activity has streaming input parameters, then activity parameter node listeners (see 8.9.2.7) are registered for the activity parameter nodes corresponding to those parameters. Values posted to a streaming input parameter will be passed to the corresponding activity parameter node via the listener. An activity with streaming input parameters can continue execution even after all threads within it have (temporarily) ended, because additional values posted to those input parameters can trigger new threads within the activity execution. The execution of such an activity only terminates if this is forced by an activity final node or the raising of an exception, or if the execution is explicitly terminated by the invoker of the activity.

If an activity has streaming output parameters, then the values of any tokens accepted by the activity parameter nodes of streaming output parameters are immediately posted to those output parameters and passed on to their listeners. Note that an activity whose only streaming parameters are output parameters *cannot* continue execution once all threads within it have ended. This is because the invocation of the activity is synchronous, so, if there are no streaming input parameters, once the initial execution returns, there is no way to trigger new threads within the activity execution.

Exception Handling

Any executable node in an activity may be *protected* by one or more exception handlers. If an exception is raised during the firing of a protected executable node, then the exception value is checked to see if it is an instance of any of the exception types of one of the exception handlers for the node. If so, then the exception value is offered to the body of the handler (which must also be an executable node) on exception input object node. (See the UML 2 Specification, subclause 15.5.3.2.)

The semantics of exception handling is captured in the ExecutableNodeActivation class. However, other than this, executable nodes do not have any specific execution semantics. The only concrete executable nodes in UML are actions. Therefore, the further action-specific semantics of exception handling is covered in ActionActivation (see 8.10.2.4).

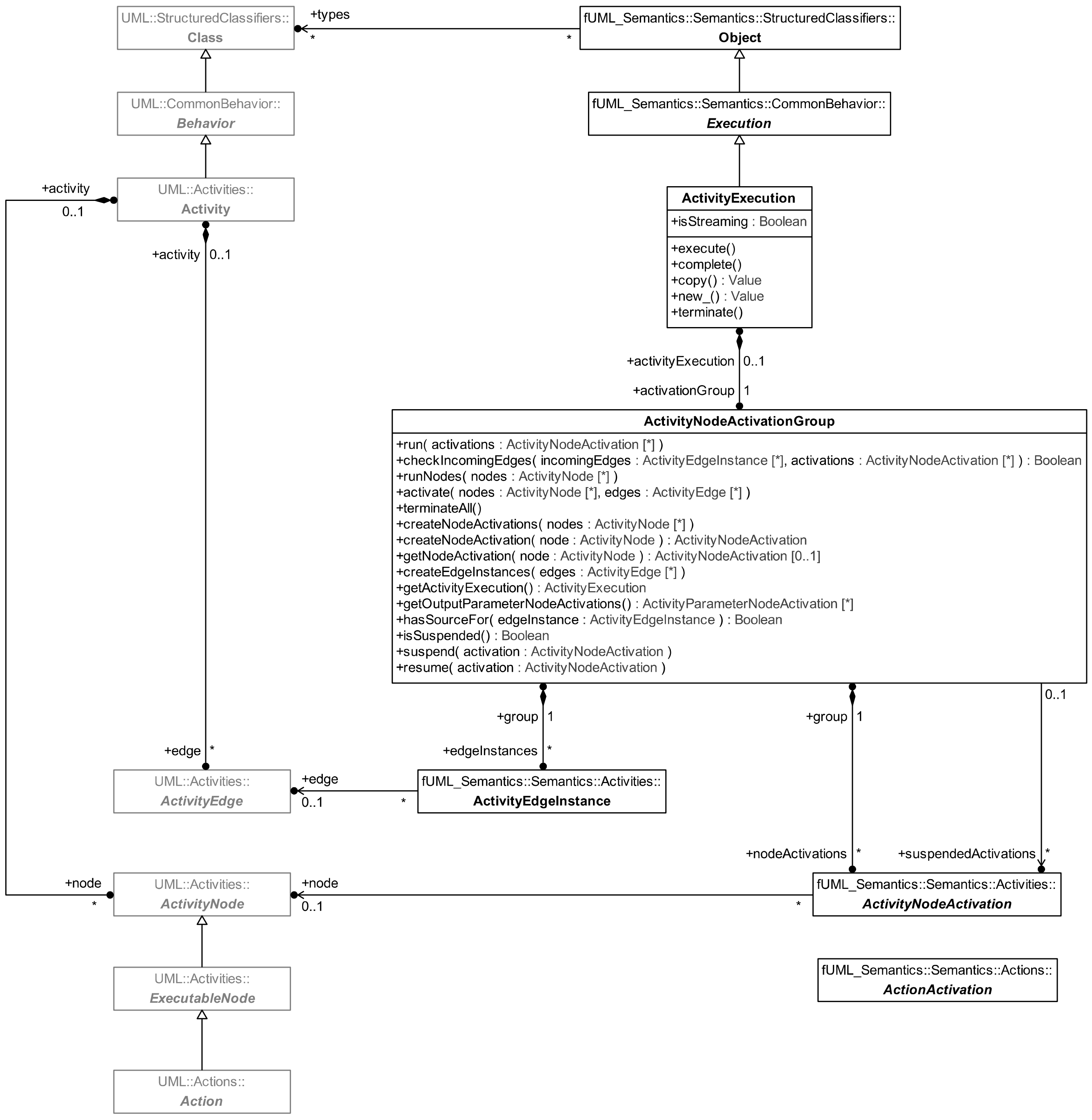

UML::StructuredClassifiers::
Class
+types
*
*
fUML_Semantics::Semantics::StructuredClassifiers::
Object
UML::CommonBehavior::
Behavior
fUML_Semantics::Semantics::CommonBehavior::
Execution
+activity
0..1
UML::Activities::
Activity
ActivityExecution
+isStreaming : Boolean
+execute()
+complete()
+copy() : Value
+new_() : Value
+terminate()
+activity
0..1
+activityExecution
0..1
+activationGroup
1
ActivityNodeActivationGroup
+run(activations : ActivityNodeActivation [*])
+checkIncomingEdges(incomingEdges : ActivityEdgeInstance [*], activations : ActivityNodeActivation [*]) : Boolean
+runNodes(nodes : ActivityNode [*])
+activate(nodes : ActivityNode [*], edges : ActivityEdge [*])
+terminateAll()
+createNodeActivations(nodes : ActivityNode [*])
+createNodeActivation(node : ActivityNode) : ActivityNodeActivation
+getNodeActivation(node : ActivityNode) : ActivityNodeActivation [0..1]
+createEdgeInstances(edges : ActivityEdge [*])
+getActivityExecution() : ActivityExecution
+getOutputParameterNodeActivations() : ActivityParameterNodeActivation [*]
+hasSourceFor(edgeInstance : ActivityEdgeInstance) : Boolean
+isSuspended() : Boolean
+suspend(activation : ActivityNodeActivation)
+resume(activation : ActivityNodeActivation)
+group
1
+group
1
0..1
+edge
*
+edgeInstances
*
UML::Activities::
ActivityEdge
+edge
0..1
*
fUML_Semantics::Semantics::Activities::
ActivityEdgeInstance
+nodeActivations
*
+suspendedActivations
*
+node
*
UML::Activities::
ActivityNode
+node
0..1
*
fUML_Semantics::Semantics::Activities::
ActivityNodeActivation
fUML_Semantics::Semantics::Actions::
ActionActivation
UML::Activities::
ExecutableNode
UML::Actions::
Action

Figure 8.24 - Activity Executions

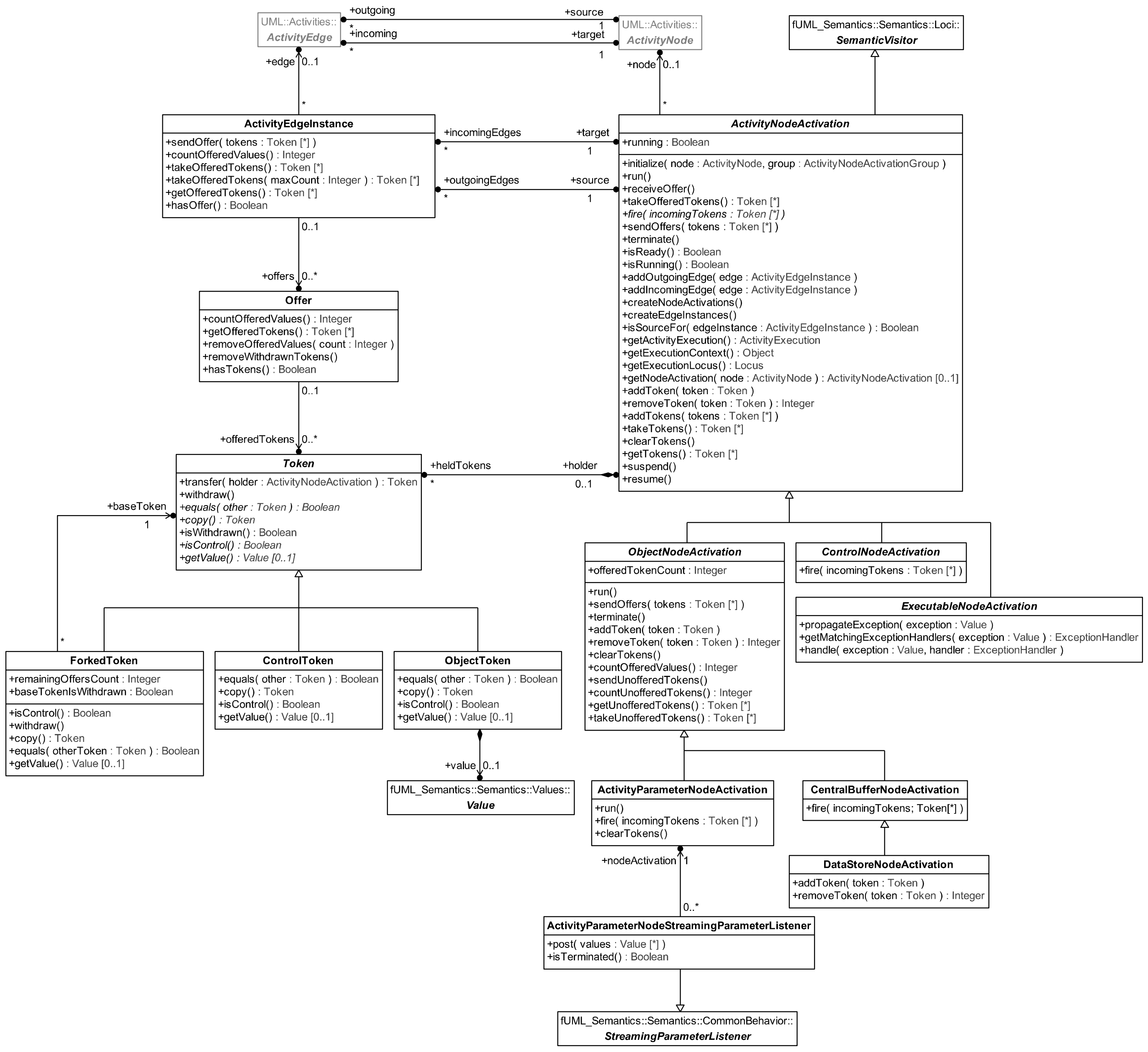
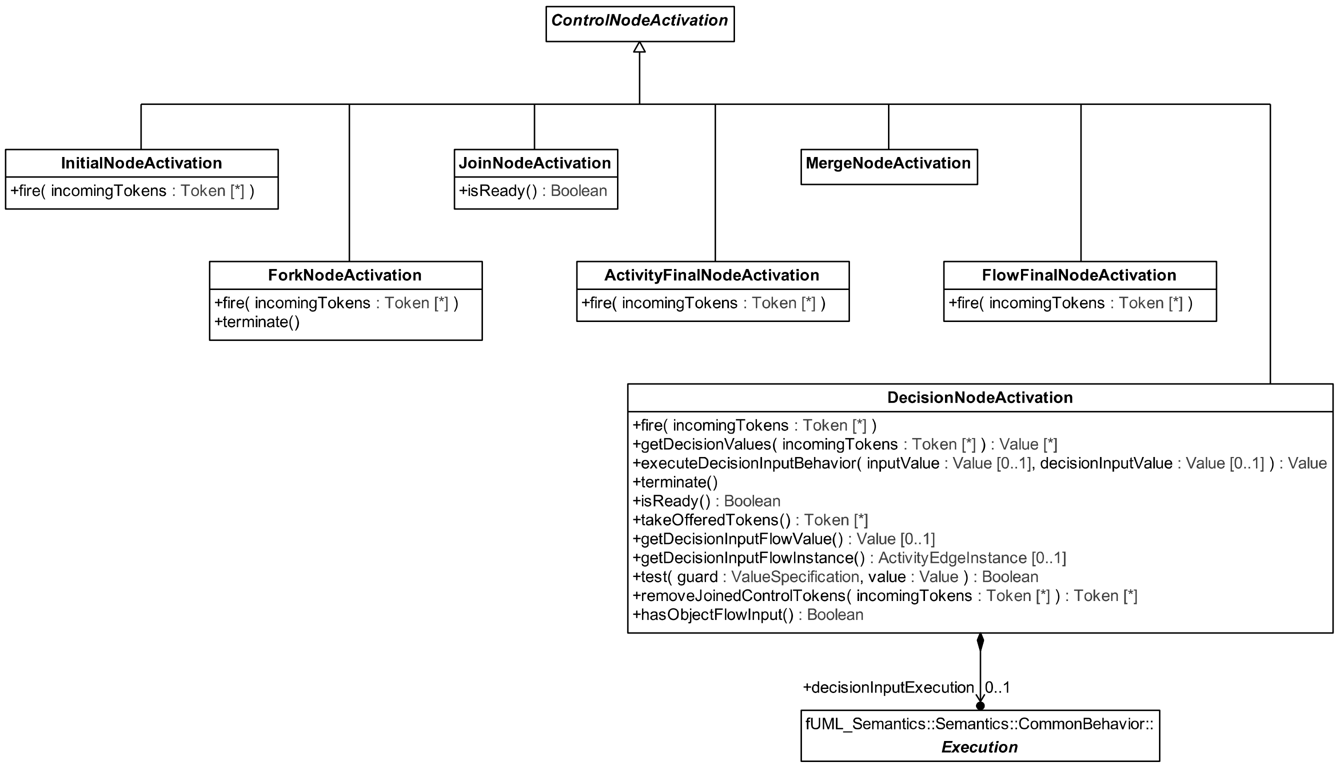

UML::Activities::
ActivityEdge
+outgoing
+incoming
+source
+target
UML::Activities::
ActivityNode
fUML_Semantics::Semantics::Loci::
SemanticVisitor
+edge 0..1
+node 0..1
ActivityEdgeInstance
+sendOffer(tokens : Token [*])
+countOfferedValues() : Integer
+takeOfferedTokens() : Token [*]
+takeOfferedTokens(maxCount : Integer) : Token [*]
+getOfferedTokens() : Token [*]
+hasOffer() : Boolean
+incomingEdges
+target
+outgoingEdges
+source
ActivityNodeActivation
+running : Boolean
+initialize(node : ActivityNode, group : ActivityNodeActivationGroup)
+run()
+receiveOffer()
+takeOfferedTokens() : Token [*]
+fire(incomingTokens : Token [*])
+sendOffers(tokens : Token [*])
+terminate()
+isReady() : Boolean
+isRunning() : Boolean
+addOutgoingEdge(edge : ActivityEdgeInstance)
+addIncomingEdge(edge : ActivityEdgeInstance)
+createNodeActivations()
+createEdgeInstances()
+isSourceFor(edgeInstance : ActivityEdgeInstance) : Boolean
+getActivityExecution() : ActivityExecution
+getExecutionContext() : Object
+getExecutionLocus() : Locus
+getNodeActivation(node : ActivityNode) : ActivityNodeActivation [0..1]
+addToken(token : Token)
+removeToken(token : Token) : Integer
+addTokens(tokens : Token [*])
+takeTokens() : Token [*]
+clearTokens()
+getTokens() : Token [*]
+suspend()
+resume()
+offers 0..*
Offer
+countOfferedValues() : Integer
+getOfferedTokens() : Token [*]
+removeOfferedValues(count : Integer)
+removeWithdrawnTokens()
+hasTokens() : Boolean
+offeredTokens 0..*
Token
+transfer(holder : ActivityNodeActivation) : Token
+withdraw()
+equals(other : Token) : Boolean
+copy() : Token
+isWithdrawn() : Boolean
+isControl() : Boolean
+getValue() : Value [0..1]
+heldTokens
+holder 0..1
+baseToken 1
ObjectNodeActivation
+offeredTokenCount : Integer
+run()
+sendOffers(tokens : Token [*])
+terminate()
+addToken(token : Token)
+removeToken(token : Token) : Integer
+clearTokens()
+countOfferedValues() : Integer
+sendUnofferedTokens()
+countUnofferedTokens() : Integer
+getUnofferedTokens() : Token [*]
+takeUnofferedTokens() : Token [*]
ControlNodeActivation
+fire(incomingTokens : Token [*])
ExecutableNodeActivation
+propagateException(exception : Value)
+getMatchingExceptionHandlers(exception : Value) : ExceptionHandler
+handle(exception : Value, handler : ExceptionHandler)
ForkedToken
+remainingOffersCount : Integer
+baseTokenIsWithdrawn : Boolean
+isControl() : Boolean
+withdraw()
+copy() : Token
+equals(otherToken : Token) : Boolean
+getValue() : Value [0..1]
ControlToken
+equals(other : Token) : Boolean
+copy() : Token
+isControl() : Boolean
+getValue() : Value [0..1]
ObjectToken
+equals(other : Token) : Boolean
+copy() : Token
+isControl() : Boolean
+getValue() : Value [0..1]
+value 0..1
fUML_Semantics::Semantics::Values::
Value
ActivityParameterNodeActivation
+run()
+fire(incomingTokens : Token [*])
+clearTokens()
CentralBufferNodeActivation
+fire(incomingTokens; Token[*])
DataStoreNodeActivation
+addToken(token : Token)
+removeToken(token : Token) : Integer
+nodeActivation 1
0..*
ActivityParameterNodeStreamingParameterListener
+post(values : Value [*])
+isTerminated() : Boolean
fUML_Semantics::Semantics::CommonBehavior::
StreamingParameterListener

Figure 8.25 - Node Activations



ControlNodeActivation
InitialNodeActivation
+fire(incomingTokens : Token [*])
JoinNodeActivation
+isReady() : Boolean
MergeNodeActivation
ForkNodeActivation
+fire(incomingTokens : Token [*])
+terminate()
ActivityFinalNodeActivation
+fire(incomingTokens : Token [*])
FlowFinalNodeActivation
+fire(incomingTokens : Token [*])
DecisionNodeActivation
+fire(incomingTokens : Token [*])
+getDecisionValues(incomingTokens : Token [*]) : Value [*]
+executeDecisionInputBehavior(inputValue : Value [0..1], decisionInputValue : Value [0..1]) : Value
+terminate()
+isReady() : Boolean
+takeOfferedTokens() : Token [*]
+getDecisionInputFlowValue() : Value [0..1]
+getDecisionInputFlowInstance() : ActivityEdgeInstance [0..1]
+test(guard : ValueSpecification, value : Value) : Boolean
+removeJoinedControlTokens(incomingTokens : Token [*]) : Token [*]
+hasObjectFlowInput() : Boolean
+decisionInputExecution
0..1
fUML_Semantics::Semantics::CommonBehavior::
Execution

Figure 8.26 - Control Node Activations

8.9.2 Class Descriptions

8.9.2.1 ActivityEdgeInstance

An activity edge instance is a connection between activity node activations corresponding to an edge between the corresponding nodes of those activations

Generalizations

None

Attributes

None

Associations

- edge : ActivityEdge [0..1]
  The activity edge of which this is an instance.
  [This is optional to allow for an implicit fork node execution to be connected to its action execution by an edge instance which does not have a corresponding node in the model.]
- group : ActivityNodeActivationGroup
  The activity group that contains this activity edge instance.

- offers : Offer [0..*]
- source : ActivityNodeActivation
  The source of this activity edge instance.
  The node of the source must be the same as the source of the edge of this edge instance.
- target : ActivityNodeActivation
  The target of this activity edge instance.
  The node of the target must be the same as the target of the edge of this edge instance.

Operations

[1] countOfferedValues ( ) : Integer

```
// Return the number of values being offered in object tokens.

int count = 0;
OfferList offers = this.offers;
for (int i = 0; i < offers.size(); i++) {
     count = count + offers.getValue(i).countOfferedValues();
}

return count;
```

[2] getOfferedTokens ( ) : Token [0..*]

```
// Get the offered tokens (after which the tokens will still be offered).

TokenList tokens = new TokenList();
OfferList offers = this.offers;

for (int i = 0; i < offers.size(); i++) {
     TokenList offeredTokens = offers.getValue(i).getOfferedTokens();
     for (int j = 0; j < offeredTokens.size(); j++) {
          tokens.addValue(offeredTokens.getValue(j));
     }
}

return tokens;
```

[3] hasOffer ( ) : Boolean

```
// Return true if there are any pending offers.

boolean hasTokens = false;
int i = 1;
```

```
while (!hasTokens & i <= this.offers.size()) {
     hasTokens = this.offers.getValue(i-1).hasTokens();
    i = i + 1;
}

return hasTokens;
```

[4] sendOffer (in tokens : Token [0..*])

```
// Send an offer  from the source to the target.
// Keep the offered tokens until taken by the target.
// (Note that any one edge should only be handling either all object tokens or all control
tokens.)

Offer offer = new Offer();

for (int i = 0; i < tokens.size(); i++) {
     Token token = tokens.getValue(i);
     offer.offeredTokens.addValue(token);
}

this.offers.addValue(offer);

this.target.receiveOffer();
```

[5] takeOfferedTokens ( ) : Token [0..*]

```
// Take all the offered tokens and return them.

TokenList tokens = new TokenList();

while (this.offers.size() > 0) {
     TokenList offeredTokens = this.offers.getValue(0).getOfferedTokens();
     for (int i = 0; i < offeredTokens.size(); i++) {
          tokens.addValue(offeredTokens.getValue(i));
     }
     this.offers.removeValue(0);
}

return tokens;
```

[6] takeOfferedTokens (in maxCount : Integer) : Token [0..*]

```
// Take all the offered tokens, up to the given maximum count of non-null object tokens, and
return them.

TokenList tokens = new TokenList();
int remainingCount = maxCount;

while (this.offers.size() > 0 & remainingCount > 0) {
    Offer offer = this.offers.getValue(0);
    TokenList offeredTokens = offer.getOfferedTokens();
    int count = offer.countOfferedValues();
    if (count <= remainingCount) {
        for (int i = 0; i < offeredTokens.size(); i++) {
            tokens.addValue(offeredTokens.getValue(i));
        }
        remainingCount = remainingCount - count;
        this.offers.removeValue(0);
    } else {
        for (int i = 0; i < remainingCount; i++) {
            Token token = offeredTokens.getValue(i);
            if (token.getValue() != null) {
                tokens.addValue(token);
            }
        }
        offer.removeOfferedValues(remainingCount);
        remainingCount = 0;
    }
}

return tokens;
```

8.9.2.2 ActivityExecution

An activity execution is used to execute a specific activity. The type of the activity execution must be an activity.

When executed, the activity execution creates activity edge instances for all activity edges, activity node activations for all activity nodes and makes offers to all nodes with no incoming edges.

Execution terminates when either all node activations are complete, or an activity final node is executed.

Generalizations

- Execution

Attributes

isStreaming
Whether the activity being executed has streaming input parameters.

Associations

- activationGroup : ActivityNodeActivationGroup
  The group of activations of the activity nodes of the activity.

Operations

[1] complete ( )

```
// Copy the values on the tokens offered by output parameter nodes for
// non-stream parameters to the corresponding output parameter values.

Activity activity = (Activity) (this.getTypes().getValue(0));

ActivityParameterNodeActivationList outputActivations = this.activationGroup
        .getOutputParameterNodeActivations();

for (int i = 0; i < outputActivations.size(); i++) {
    ActivityParameterNodeActivation outputActivation = outputActivations
            .getValue(i);

    Parameter parameter = ((ActivityParameterNode) (outputActivation.node)).parameter;

    if (!parameter.isStream) {
        ParameterValue parameterValue = new ParameterValue();
        parameterValue.parameter = parameter;

        TokenList tokens = outputActivation.getTokens();
        for (int j = 0; j < tokens.size(); j++) {
            Token token = tokens.getValue(j);
            Value value = ((ObjectToken) token).value;
            if (value != null) {
                parameterValue.values.addValue(value);
            }
        }

        this.setParameterValue(parameterValue);
    }
}
```

[2] copy ( ) : Value

```
// Create a new activity execution that is a copy of this execution.
// [Note: This currently just returns a non-executing execution for the same activity as this
execution.]

return super.copy();
```

[3] execute ( )

// Execute the activity for this execution by creating an activity node

```
// activation group and activating all the activity nodes in the
// activity. If the activity has no streaming input parameters, then, when
// the execution is complete, copy the values on the tokens offered by
// output parameter nodes to the corresponding output parameters.

Activity activity = (Activity) (this.getTypes().getValue(0));

this.isStreaming = false;
int i = 1;
ParameterList parameters = activity.ownedParameter;
while (i <= parameters.size() & !this.isStreaming) {
    Parameter parameter = parameters.getValue(i - 1);
    this.isStreaming =
        (parameter.direction == ParameterDirectionKind.in |
         parameter.direction == ParameterDirectionKind.inout) &
        parameter.isStream;
    i = i + 1;
}

this.activationGroup = new ActivityNodeActivationGroup();
this.activationGroup.activityExecution = this;
this.activationGroup.activate(activity.node, activity.edge);

if (!this.isStreaming) {
    this.complete();
}
```

[4] new_ ( ) : Value

```
// Create a new activity execution with empty properties.

return new ActivityExecution();
```

[5] terminate ( )

```
// Terminate all node activations. If this execution is non-streaming,
// then this is sufficient to result in the activity execution ultimately
// completing. Otherwise, explicitly complete the execution.

if (this.activationGroup != null) {
    this.activationGroup.terminateAll();
}

if (this.isStreaming) {
    this.complete();
}
```

8.9.2.3 ActivityFinalNodeActivation

An activity final node activation is a control node activation for a node that is an activity final node.

Generalizations

- ControlNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// Terminate the activity execution or structured node activation
// containing this activation.

if (incomingTokens.size() > 0 | this.incomingSize() == 0) {
      if (this.group.activityExecution !=null) {
            this.group.activityExecution.terminate();
      } else if (this.group.containingNodeActivation != null) {
            this.group.containingNodeActivation.terminateAll();
      } else if (this.group instanceof ExpansionActivationGroup) {
            ((ExpansionActivationGroup)this.group).regionActivation.terminate();
```

```
        }
}
```

8.9.2.4 ActivityNodeActivation

An activity node activation is used to define the behavior of an activity node in the context of a containing activity or structured activity node.

Generalizations

- SemanticVisitor

Attributes

- running : Boolean
  If true, this node activation is enabled for execution once all its other prerequisites are satisfied.

Associations

- group : ActivityNodeActivationGroup
  The group that contains this activity node activation.
- heldTokens : Token [0..*]
- incomingEdges : ActivityEdgeInstance [0..*]
  The set of activity edge instances for the incoming edges of the node.
- node : ActivityNode [0..1]
  The activity node being activated by this activity node activation. The node must be owned by the activity (type) of the activity execution of this node activation.
  [This is optional, to allow for fork node edge queues and implicit fork and join node activations for actions to not have nodes in the model.]
- outgoingEdges : ActivityEdgeInstance [0..*]
  The set of activity edge instances for the outgoing edges of the node.

Operations

[1] addIncomingEdge (in edge : ActivityEdgeInstance)

```
// Add an activity edge instance as an incoming edge of this activity node activation.

edge.target = this;
this.incomingEdges.addValue(edge);
```

[2] addOutgoingEdge (in edge : ActivityEdgeInstance)

```
// Add an activity edge instance as an outgoing edge of this activity node activation.

edge.source = this;
this.outgoingEdges.addValue(edge);
```

[3] addToken (in token : Token)

```
// Transfer the given token to be held by this node.

Token transferredToken = token.transfer(this);
this.heldTokens.addValue(transferredToken);
```

[4] addTokens (in tokens : Token [0..*])

```
// Transfer the given tokens to be the held tokens for this node.

for (int i = 0; i < tokens.size(); i++) {
     Token token = tokens.getValue(i);
     this.addToken(token);
}
```

[5] clearTokens ( )

```
// Remove all held tokens.

while (this.heldTokens.size() > 0) {
     this.heldTokens.getValue(0).withdraw();
}
```

[6] createEdgeInstances ( )

```
// Create edge instances for any edge instances owned by the node for this activation.
// For most kinds of nodes, this does nothing.

return;
```

[7] createNodeActivations ( )

```
// Create node activations for any subnodes of the node for this activation.
// For most kinds of nodes, this does nothing.

return;
```

[8] fire (in incomingTokens : Token [0..*])

```
Carry out the main behavior of this activity node.
```

[9] getActivityExecution ( ) : ActivityExecution

```
// Return the activity execution that contains this activity node activation, directly or
indirectly.

return  this.group.getActivityExecution();
```

[10] getExecutionContext ( ) : Object

```
// Get the context object for the containing activity execution.

return  this.getActivityExecution().context;
```

[11] getExecutionLocus ( ) : Locus

```
// Get the locus of the containing activity execution.

return  this.getActivityExecution().locus;
```

[12] getNodeActivation (in node : ActivityNode) : ActivityNodeActivation [0..1]

```
// Get the activity node activation corresponding to the given activity node, in the context
of this activity node activation.
// By default, return this activity node activation, if it is for the given node, otherwise
return nothing.

ActivityNodeActivation activation = null;
if (node == this.node) {
    activation = this;
}

return activation;
```

[13] getTokens ( ) : Token [0..*]

```
// Get the tokens held by this node activation.

TokenList tokens = new TokenList();
TokenList heldTokens = this.heldTokens;
for (int i = 0; i < heldTokens.size(); i++) {
     Token heldToken = heldTokens.getValue(i);
     tokens.addValue(heldToken);
}

return tokens;
```

[14] initialize (node : ActivityNode, group : ActivityNodeActivationGroup)

```
// Initialize this node activation.

this.node = node;
this.group = group;
this.running = false;
```

[15] isReady ( ) : Boolean

```
// Check if all the prerequisites for this node have been satisfied.
// By default, check that this node is running.

return  this.isRunning();
```

[16] isRunning ( ) : Boolean

```
// Test whether this node activation is running.

return  this.running;
```

[17] isSourceFor (in edgeInstance : ActivityEdgeInstance) : Boolean

```
// Check if this node activation is the effective source for the given edge instance.

return edgeInstance.source == this;
```

[18] receiveOffer ( )

```
// Receive an offer from an incoming edge.
// Check if all prerequisites have been satisfied. If so, fire.

_beginIsolation();

    boolean ready = this.isReady();

    TokenList tokens = new TokenList();
     if (ready) {
        tokens = this.takeOfferedTokens();
    }

_endIsolation();
```

```
if (ready) {
     this.fire(tokens);
}
```

[19] removeToken (in token : Token) : Integer

```
// Remove the given token, if it is held by this node activation.
// Return the position (counting from 1) of the removed token (0 if there is none removed).

boolean notFound = true;
int i = 1;
while (notFound & i <= this.heldTokens.size()) {
     if (this.heldTokens.getValue(i-1) == token) {
          this.heldTokens.remove(i-1);
          notFound = false;
     }
     i = i + 1;
}

if (notFound) {
    i = 0;
} else {
    i = i - 1;
}

return i ;
```

[20] resume ( )

```
// Resume this activation within the activation group that contains it.

this.group.resume(this);
```

[21] run ( )

```
// Run the activation of this node.

this.running = true;
```

[22] sendOffers (in tokens : Token [0..*])

```
// Send offers for the given set of tokens over all outgoing edges (if there are any tokens
actually being offered).
```

```
if (tokens.size()>0) {

    // *** Send all outgoing offers concurrently. ***
    ActivityEdgeInstanceList outgoingEdges = this.outgoingEdges;
    for (Iterator i = outgoingEdges.iterator(); i.hasNext();) {
        ActivityEdgeInstance outgoingEdge = (ActivityEdgeInstance)i.next();
        outgoingEdge.sendOffer(tokens);
    }

}
```

[23] suspend ( )

```
// Ssupend this activation within the activation group that contains it.

this.group.suspend(this);
```

[24] takeOfferedTokens ( ) : Token [0..*]

```
// Get tokens from all incoming edges.

TokenList allTokens = new TokenList();
ActivityEdgeInstanceList incomingEdges = this.incomingEdges;
for (int i = 0; i < incomingEdges.size(); i++) {
    ActivityEdgeInstance incomingEdge = incomingEdges.getValue(i);
    TokenList tokens = incomingEdge.takeOfferedTokens();
    for (int j = 0; j < tokens.size(); j ++) {
        Token token = tokens.getValue(j);
        allTokens.addValue(token);
    }
}

return allTokens;
```

[25] takeTokens ( ) : Token [0..*]

```
// Take the tokens held by this node activation.

TokenList tokens = this.getTokens();
this.clearTokens();

return tokens;
```

[26] terminate ( )

```
// Terminate the activation of this node.


this.running = false;
```

8.9.2.5 ActivityNodeActivationGroup

An activity node group is a group of nodes that are activated together, either directly in the context of an activity execution, or in the context of.

Generalizations

None

Attributes

None

Associations

- activityExecution : ActivityExecution [0..1]
  The activity execution to which this group belongs.
  (This will be empty if the group is for a structured activity node activation.)
- containingNodeActivation : StructuredActivityNodeActivation [0..1]
  The structured activity node activation to which this group belongs.
  (This will be empty if the group is for an activity execution.)
- edgeInstances : ActivityEdgeInstance [0..*]
  The set of activity edge instances for this group.
- nodeActivations : ActivityNodeActivation [0..*]
  The set of activity node executions for this group.
- suspendedActivations : ActivityNodeActivation [0..*]
  Activity node activations in this activation group that are suspended waiting for an event occurrence. If an activation group has a containing node activation and any suspended activations, then the containing node activation will also be suspended.

Operations

[1] activate (in nodes : ActivityNode [0..*], in edges : ActivityEdge [0..*])

```
// Activate and run the given set of nodes with the given set of edges, within this
activation group.


this.createNodeActivations(nodes);
this.createEdgeInstances(edges);
this.run(this.nodeActivations);
```

[2] checkIncomingEdges (in incomingEdges : ActivityEdgeInstance [0..*], in activations : ActivityNodeActivation [0..*]) : Boolean

```
// Check if any incoming edges have a source in a given set of activations.

int j = 1;
boolean notFound = true;

while (j <= incomingEdges.size() & notFound) {
    int k = 1;
    while (k <= activations.size() & notFound) {
        if (activations.getValue(k-1).isSourceFor(incomingEdges.getValue(j-1))) {
            notFound = false;
        }
        k = k + 1;
    }
    j = j + 1;
}

return notFound;
```

[3] createEdgeInstances (in edges : ActivityEdge [0..*])

```
// Create instance edges for the given activity edges, as well as for edge instances within
any nodes activated in this group.

for (int i = 0; i < edges.size(); i++) {
    ActivityEdge edge = edges.getValue(i);

    ActivityEdgeInstance edgeInstance = new ActivityEdgeInstance();
    edgeInstance.edge = edge;
    edgeInstance.group = this;

    this.edgeInstances.addValue(edgeInstance);
    this.getNodeActivation(edge.source).addOutgoingEdge(edgeInstance);
    this.getNodeActivation(edge.target).addIncomingEdge(edgeInstance);

}

ActivityNodeActivationList nodeActivations = this.nodeActivations;
for (int i = 0; i < nodeActivations.size(); i++) {
    ActivityNodeActivation nodeActivation = nodeActivations.getValue(i);
    nodeActivation.createEdgeInstances();
}
```

[4] createNodeActivation (in node : ActivityNode) : ActivityNodeActivation

```
// Create an activity node activation for a given activity node in this activity node
activation group.

ActivityNodeActivation activation = (ActivityNodeActivation)
(this.getActivityExecution().locus.factory.instantiateVisitor(node));
activation.initialize(node, this);

this.nodeActivations.addValue(activation);

activation.createNodeActivations();

return activation;
```

[5] createNodeActivations (in nodes : ActivityNode [0..*])

```
// Add activity node activations for the given set of nodes to this group and create edge
instances between them.

for (int i = 0; i < nodes.size(); i++) {
     ActivityNode node = nodes.getValue(i);

     this.createNodeActivation(node);

}
```

[6] getActivityExecution ( ) : ActivityExecution

```
// Return the activity execution to which this group belongs, directly or indirectly.

ActivityExecution activityExecution = this.activityExecution;
if (activityExecution == null) {
     activityExecution = this.containingNodeActivation.group.getActivityExecution();
}

return activityExecution;
```

[7] getNodeActivation (in node : ActivityNode) : ActivityNodeActivation [0..1]

```
// Return the node activation (if any) in this group,
// or any nested group, corresponding to the given activity node.
// If this is a group for a structured activity node activation,
// also include the pin activations for that node activation.
```

```
ActivityNodeActivation activation = null;

if (this.containingNodeActivation != null && node instanceof Pin) {
     activation = this.containingNodeActivation.getPinActivation((Pin)node);
}

if (activation == null) {
    int i = 1;
     while (activation == null & i <= this.nodeActivations.size()) {
          activation = this.nodeActivations.getValue(i-1).getNodeActivation(node);
         i = i + 1;
    }
}

return activation;
```

[8] getOutputParameterNodeActivations ( ) : ActivityParameterNodeActivation [0..*]

```
// Return the set of all activations in this group of activity parameter nodes for output
(inout, out and return) parameters.

ActivityParameterNodeActivationList parameterNodeActivations = new
ActivityParameterNodeActivationList();
ActivityNodeActivationList nodeActivations = this.nodeActivations;
for (int i = 0; i < nodeActivations.size(); i++) {
     ActivityNodeActivation activation = nodeActivations.getValue(i);
     if (activation instanceof ActivityParameterNodeActivation) {
          if (activation.incomingEdges.size() > 0 {
              parameterNodeActivations.addValue((ActivityParameterNodeActivation)activation);
         }
    }
}

return parameterNodeActivations;
```

[9] hasSourceFor (edgeInstance : activityEdgeInstance) : Boolean

```
// Returns true if this activation group has a node activation
// corresponding to the source of the given edge instance.

boolean hasSource = false;
```

```
ActivityNodeActivationList activations = this.nodeActivations;
int i = 1;
while !hasSource & i <= activations.size()) {
     hasSource = activations.getValue(i-1).isSourceFor(edgeInstance);
     i = i + 1;
}
return hasSource;
```

[10] isSuspended ( ) : Boolean

```
// Check if this activation group has any suspended activations and is,
// therefore, itself suspended.

return this.suspendedActivations.size() > 0;
```

[11] resume (activation : ActivityNodeActivation)

```
// Resume the given activation by removing it from the suspended
// activation list for this activation group. If this is the last
// suspended activation, and the activation group h as a containing
// node activation, then resume that containing activation.

boolean found = false;
int i = 1;
while (!found & i <= this.suspendedActivations.size()) {
     if (this.suspendedActivations.get(i-1) == activation) {
          this.suspendedActivations.removeValue(i-1);
          found = true;
     }
     i = i + 1;
}
if (!this.isSuspended()) {
     StructuredActivityNodeActivation containingNodeActivation =
this.containingNodeActivation;
     if (containingNodeActivation != null) {
          containingNodeActivation.resume();
     }
}
```

[12] run (in activations : ActivityNodeActivation [0..*])

```
// Run the given node activations.
// Then concurrently send offers to all input activity parameter node activations (if any).
// Finally, concurrently send offers to all activations of other kinds of nodes that have
// no incoming edges with the given set (if any).

for (int i = 0; i < activations.size(); i++) {
     ActivityNodeActivation activation = activations.getValue(i);
     activation.run();
}

ActivityNodeActivationList enabledParameterNodeActivations = new
ActivityNodeActivationList();
ActivityNodeActivationList enabledOtherActivations = new ActivityNodeActivationList();

for (int i = 0; i < activations.size(); i++) {
     ActivityNodeActivation activation = activations.getValue(i);

     if (!(activation instanceof PinActivation |
           activation instanceof ExpansionNode)) {
          boolean isEnabled = this.checkIncomingEdges(activation.incomingEdges, activations);

          // For an action activation, also consider incoming edges to input pins
          if (isEnabled & activation instanceof ActionActivation) {
              InputPinList inputPins = ((Action)activation.node).input;
              int j = 1;
              while (j <= inputPins.size() & isEnabled) {
                   InputPin inputPin = inputPins.getValue(j-1);
                   ActivityEdgeInstanceList inputEdges =
((ActionActivation)activation).getPinActivation(inputPin).incomingEdges;
                   isEnabled = this.checkIncomingEdges(inputEdges, activations);
                   j = j + 1;
              }
          }

          if (isEnabled) {

               if (activation instanceof ActivityParameterNodeActivation) {
                    enabledParameterNodeActivations.addValue(activation);
               } else {
                    enabledOtherActivations.addValue(activation);
               }
          }
     }
}
```

```
// *** Send offers to all enabled activity parameter nodes concurrently. ***
for (Iterator i = enabledParameterNodeActivations.iterator(); i.hasNext();) {
     ActivityNodeActivation activation = (ActivityNodeActivation) i.next();
     activation.receiveOffer();
}

// *** Send offers to all other enabled nodes concurrently. ***
for (Iterator i = enabledOtherActivations.iterator(); i.hasNext();) {
     ActivityNodeActivation activation = (ActivityNodeActivation)i.next();
     activation.receiveOffer();
}
```

[13] runNodes (in nodes : ActivityNode [0..*])

```
// Run the node activations associated with the given nodes in this activation group.

ActivityNodeActivationList nodeActivations = new ActivityNodeActivationList();

for (int i = 0; i < nodes.size(); i++) {
     ActivityNode node = nodes.getValue(i);
     ActivityNodeActivation nodeActivation = this.getNodeActivation(node);
     if (nodeActivation != null) {
          nodeActivations.addValue(nodeActivation);
     }
}

this.run(nodeActivations);
```

[14] suspend (activation : ActivityNodeActivation)

```
// Suspend the given activation in this activation group. If this is
// the only suspended activation, and the activation group has a
// containing node activation, then suspend that containing activation.

if (!this.isSuspended()) {
     StructuredActivityNodeActivation containingNodeActivation =
this.containingNodeActivation;
     if (containingNodeActivation != null) {
          containingNodeActivation.suspend();
     }
}
this.suspendedActivations.addValue(activation);
```

[15] terminateAll ( )

```
// Terminate all node activations in the group.

ActivityNodeActivationList nodeActivations = this.nodeActivations;
for (int i = 0; i < nodeActivations.size(); i++) {
     ActivityNodeActivation nodeActivation = nodeActivations.getValue(i);
     nodeActivation.terminate();
}
```

8.9.2.6 ActivityParameterNodeActivation

An activity parameter node activation is an object node activation for a node that is an activity parameter node.

Generalizations

- ObjectNodeActivation

Attributes

None

Associations

None

Operations

[1] clearTokens ( )

```
// Clear all held tokens only if this is an input parameter node.

if (this.node.incoming.size() == 0) {
     super.clearTokens();
}
```

[2] fire (in incomingTokens : Token [0..*])

```
// If there are no incoming edges, this is an activation of an input
// activity parameter node.
// Get the values from the input parameter indicated by the activity
// parameter node and offer those values as object tokens.
// If there are one or more incoming edges, this is an activation of an
// output activity parameter node.
// If the output parameter is not streaming, take the tokens offered on
// incoming edges and add them to the set of tokens being offered.
// If the output parameter is streaming, post the values from the
// the tokens offered on incoming edges to the parameter value.
// (Note that an output activity parameter node may fire multiple times,
// accumulating tokens offered to it.)

Parameter parameter = ((ActivityParameterNode) (this.node)).parameter;
ParameterValue parameterValue = this.getActivityExecution().
          getParameterValue(parameter);

if (this.node.incoming.size() == 0) {
    if (parameterValue != null) {
        ValueList values = parameterValue.values;
        for (int i = 0; i < values.size(); i++) {
            Value value = values.getValue(i);
            ObjectToken token = new ObjectToken();
            token.value = value;
            this.addToken(token);
        }
        this.sendUnofferedTokens();
    }
}

else {
    this.addTokens(incomingTokens);

    if (parameterValue instanceof StreamingParameterValue) {
        ValueList values = new ValueList();
        for (int i = 0; i < incomingTokens.size(); i++) {
            Token token = incomingTokens.getValue(i);
            Value value = token.getValue();
            if (value != null) {
                values.addValue(value);
            }
        }
        ((StreamingParameterValue)parameterValue).post(values);
        super.clearTokens();
    }
}
```

[3] run ( )

```
// If this activation is for an input activity parameter node for a
// stream parameter, then register a listener for this activation
// with the streaming parameter value corresponding to the parameter.

super.run();

Parameter parameter = ((ActivityParameterNode) (this.node)).parameter;
ParameterValue parameterValue = this.getActivityExecution().
          getParameterValue(parameter);
if (this.node.incoming.size() == 0 &
        parameterValue instanceof StreamingParameterValue) {
    ActivityParameterNodeStreamingParameterListener listener =
            new ActivityParameterNodeStreamingParameterListener();
    listener.nodeActivation = this;
   ((StreamingParameterValue)parameterValue).register(listener);
}
```

8.9.2.7 ActivityParameterNodeStreamingParameterListener

An activity parameter node streaming parameter listener is a streaming parameter listener for posting values from a streaming parameter value to an activity parameter node (which is presumed to be for an input parameter).

Generalizations

- StreamingParameterListener

Attributes

None

Associations

- nodeActivation : ActivityParameterNodeActivation
  The node activation for the activity parameter node to which streaming parameter values are to be posted.

Operations

[1] isTerminated ( ) : Boolean

```
// This listener is terminated if the node activation is not running.

return !this.nodeActivation.isRunning();
```

[2] post (values : Value [0..*])

```
// Fire the activity parameter node activation.
// (Note that the values do not have to be passed to the node activation,
// because an input activity parameter node activation retrieves values
// directly from the relevant parameter value.)
```

```
nodeActivation.fire(new TokenList());
```

8.9.2.8 CentralBufferNodeActivation

A central buffer node activation is an object node activation for a node that is a central buffer node.

Generalizations

- ObjectNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// Add all incoming tokens to the central buffer node.
// Offer any tokens that have not yet been offered.

this.addTokens(incomingTokens);
this.sendUnofferedTokens();
```

8.9.2.9 ControlNodeActivation

A control node activation is an activity node activation for a node that is a control node.

Generalizations

- ActivityNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// By default, offer all tokens on all outgoing edges.

this.sendOffers(incomingTokens);
```

8.9.2.10 ControlToken

A control token represents the passing of control along a control flow edge.

Generalizations

- Token

Attributes

None

Associations

None

Operations

[1] copy ( ) : Token

```
// Return a new control token.

return new ControlToken();
```

[2] equals (in other : Token) : Boolean

```
// Return true if the other token is a control token, because control tokens are
interchangeable.

return other instanceof ControlToken;
```

[3] getValue ( ) : Value [0..1]

```
// Control tokens do not have values.

return null;

[4] isControl ( ) : Boolean

// Return true for a control token.

return true;
```

8.9.2.11 DataStoreNodeActivation

A data store node activation is a central buffer node activation for a node that is a data store node.

Generalization

- CentralBufferNodeActivation

Attributes

None

Associations

None

Operations

[1] addToken (in token : Token)

```
// Add the given token to the data store only if it is unique,
// that is, if its value is not the same as the value of
// another token already held in the data store.

Value value = token.getValue();

boolean isUnique = true;
if (value != null) {
     TokenList heldTokens = this.getTokens();
     int i = 1;
     while (isUnique & i <= heldTokens.size()) {
          isUnique = !heldTokens.getValue(i-1).getValue().equals(value);
          i = i + 1;
     }
}

if (isUnique) {
     super.addToken(token);
}
```

[2] removeToken (in token : Token) : Integer

```
// Remove the given token from the data store, but then immediately
// add a copy back into the data store and offer it (unless the
// node activation has already been terminated).

int i = super.removeToken(token);

if (this.isRunning()) {
     super.addToken(token.copy());
     this.sendUnofferedTokens();
}

return i;
```

8.9.2.12 DecisionNodeActivation

Generalizations

- ControlNodeActivation

Attributes

None

Associations

- decisionInputExecution : Execution [0..1]
  The current execution of the decision input behavior (if any).

Operations

[1] executeDecisionInputBehavior (in inputValue : Value [0..1], in decisionInputValue : Value [0..1]) : Value

```
// Create the decision input execution from the decision input behavior.
// If the behavior has input parameter(s), set the input parameter(s) of the execution to the
given value(s).
// Execute the decision input execution and then remove it.
// Return the value of the output parameter of the execution.
// If there is no decision input behavior, the decision input value is returned, if one is
given, otherwise the input value is used as the decision value.

Behavior decisionInputBehavior = ((DecisionNode)(this.node)).decisionInput;

Value decisionInputResult = null;

if (decisionInputBehavior == null) {

     if (decisionInputValue != null) {
          decisionInputResult = decisionInputValue;
     } else {
          decisionInputResult = inputValue;
     }

} else {

     this.decisionInputExecution =
this.getExecutionLocus().factory.createExecution(decisionInputBehavior,
this.getExecutionContext());

     int i = 1;
     int j = 0;
     while ((j == 0 | (j == 1 & decisionInputValue != null)) & i <=
decisionInputBehavior.ownedParameter.size()) {
          Parameter parameter = decisionInputBehavior.ownedParameter.getValue(i-1);
          if (parameter.direction.equals(ParameterDirectionKind.in) |
              parameter.direction.equals(ParameterDirectionKind.inout)) {
```

```
            ParameterValue inputParameterValue = new ParameterValue();
            inputParameterValue.parameter = parameter;

            j = j +1;
            if (j == 1 && inputValue != null) {
                inputParameterValue.values.addValue(inputValue);
            } else {
                inputParameterValue.values.addValue(decisionInputValue);
            }

            this.decisionInputExecution.setParameterValue(inputParameterValue);

        }
        i = i + 1;
    }

     this.decisionInputExecution.execute();
     ParameterValueList outputParameterValues =
this.decisionInputExecution.getOutputParameterValues();
     decisionInputExecution.destroy();

     decisionInputResult = outputParameterValues.getValue(0).values.getValue(0);
}

return  decisionInputResult;
```

[2] fire (in incomingTokens : Token [0..*])

```
// Get the decision values and test them on each guard.
// Forward the offer over the edges for which the test succeeds.

//TokenList  incomingTokens  =  this.takeOfferedTokens();
TokenList  removedControlTokens  =  this.removeJoinedControlTokens(incomingTokens);
ValueList  decisionValues  =  this.getDecisionValues(incomingTokens);

ActivityEdgeInstanceList  outgoingEdges  =  this.outgoingEdges;
for  (int  i  =  0;  i  <  outgoingEdges.size();  i++)  {
     ActivityEdgeInstance  edgeInstance  =  outgoingEdges.getValue(i);
     ValueSpecification guard = edgeInstance.edge.guard;

     TokenList offeredTokens = new TokenList();
     for  (int  j  =  0;  j  <  incomingTokens.size();  j++)  {
          Token incomingToken = incomingTokens.getValue(j);
          Value decisionValue = decisionValues.getValue(j);
          if (this.test(guard, decisionValue)) {
              offeredTokens.addValue(incomingToken);
          }
     }

     if (offeredTokens.size() > 0) {
          for (int j = 0; j < removedControlTokens.size(); j++) {
               Token removedControlToken = removedControlTokens.getValue(j);
               offeredTokens.addValue(removedControlToken);
          }
          edgeInstance.sendOffer(offeredTokens);
     }
}
```

[3] getDecisionInputFlowInstance ( ) : ActivityEdgeInstance [0..1]

```
// Get the activity edge instance for the decision input flow, if any.

ActivityEdge decisionInputFlow = ((DecisionNode)(this.node)).decisionInputFlow;

ActivityEdgeInstance edgeInstance = null;
if (decisionInputFlow != null) {
     int i = 1;
     while (edgeInstance == null & i <=this.incomingEdges.size()) {
          ActivityEdgeInstance incomingEdge = this.incomingEdges.getValue(i-1);
          if (incomingEdge.edge == decisionInputFlow) {
              edgeInstance = incomingEdge;
          }
          i = i + 1;
     }
}

return edgeInstance;
```

[4] getDecisionInputFlowValue ( ) : Value [0..1]

```
// Take the next token available on the decision input flow, if any, and return its value.

ActivityEdgeInstance decisionInputFlowInstance = this.getDecisionInputFlowInstance();
```

```
Value value = null;
if (decisionInputFlowInstance != null) {
     TokenList tokens = decisionInputFlowInstance.takeOfferedTokens();
     if (tokens.size() > 0) {
          value = tokens.getValue(0).getValue();
     }
}

return value;
```

[5] getDecisionValues (in incomingTokens : Token [0..*]) : Value [0..*]

```
// If there is neither a decision input flow nor a decision input behavior, then return the set of values from the incoming tokens.
//         [In this case, the single incoming edge must be an object flow.]
// If there is a decision input flow, but no decision input behavior, then return a list of the decision input values equal in size to the number of incoming tokens.
// If there is both a decision input flow and a decision input behavior, then execute the decision input behavior once for each incoming token and return the set of resulting values.
//      If the primary incoming edge is an object flow, then the value on each object token is passed to the decision input behavior, along with the decision input flow value, if any.
//      If the primary incoming edge is a control flow, then the decision input behavior only receives the decision input flow, if any.


Value decisionInputValue = this.getDecisionInputFlowValue();

ValueList decisionValues = new ValueList();
for (int i = 0; i < incomingTokens.size(); i++) {
     Token incomingToken = incomingTokens.getValue(i);
     Value value = this.executeDecisionInputBehavior(incomingToken.getValue(), decisionInputValue);
     decisionValues.addValue(value);
}

for (int i = 0; i < decisionValues.size(); i++) {
     Value decisionValue = decisionValues.getValue(i);
}

return decisionValues;
```

[6] hasObjectFlowInput ( ) : Boolean

```
// Check that the primary incoming edge is an object flow.
```

```
ActivityEdge decisionInputFlow = ((DecisionNode)(this.node)).decisionInputFlow;

boolean isObjectFlow = false;
int i = 1;
while (!isObjectFlow & i <= this.incomingEdges.size()) {
     ActivityEdge edge = this.incomingEdges.getValue(i-1).edge;
     isObjectFlow = edge != decisionInputFlow & edge instanceof ObjectFlow;
     i = i + 1;
}

return isObjectFlow;
```

[7] isReady ( ) : Boolean

```
// Check that all incoming edges have sources that are offering tokens.
// [This should be at most two incoming edges, if there is a decision input flow.]

int i = 1;
boolean ready = true;
while (ready & i <= this.incomingEdges.size()) {
     ready = this.incomingEdges.getValue(i-1).hasOffer();
     i = i + 1;
}

return ready;
```

[8] removeJoinedControlTokens (in incomingTokens : Token [0..*]) : Token [0..*]

```
// If the primary incoming edge is an object flow, then remove any control tokens from the
incoming tokens and return them.
// [Control tokens may effectively be offered on an object flow outgoing from a join node
that has both control and object flows incoming.]

TokenList removedControlTokens = new TokenList();

if (this.hasObjectFlowInput()) {
     int i = 1;
     while (i <= incomingTokens.size()) {
          Token token = incomingTokens.getValue(i-1);
          if (token.isControl()) {
               removedControlTokens.addValue(token);
```

```
            incomingTokens.removeValue(i-1);
            i = i - 1;
        }
        i = i + 1;
    }
}

return removedControlTokens;
```

[9] takeOfferedTokens ( ) : Token [0..*]

```
// Get tokens from the incoming edge that is not the decision input flow.

ObjectFlow decisionInputFlow = ((DecisionNode)(this.node)).decisionInputFlow;

TokenList allTokens = new TokenList();
ActivityEdgeInstanceList incomingEdges = this.incomingEdges;
for (int i = 0; i < incomingEdges.size(); i++) {
    ActivityEdgeInstance edgeInstance = incomingEdges.getValue(i);
    if (edgeInstance.edge != decisionInputFlow) {
        TokenList tokens = edgeInstance.takeOfferedTokens();
        for (int j = 0; j < tokens.size(); j++) {
            allTokens.addValue(tokens.getValue(j));
        }
    }
}

return allTokens;
```

[10] terminate ( )

```
// Terminate the decision input execution, if any, and then terminate this activation.

if (this.decisionInputExecution != null) {
    this.decisionInputExecution.terminate();
}

super.terminate();
```

[11] test (in guard : ValueSpecification, in value : Value) : Boolean

```
// Test if the given value matches the guard. If there is no guard, return true.
```

```
boolean guardResult = true;
if (guard != null) {
     Value guardValue = this.getExecutionLocus().executor.evaluate(guard);
     guardResult = guardValue.equals(value);
}
return guardResult;
```

8.9.2.13 ExecutableNodeActivation

An executable node activation is an activity node activation for a node that is an executable node. It includes the specification of the semantics for the handling of exceptions raised by an executable node protected by one or more exception handlers. All other executable node semantics are covered by specializations of ExecutableNodeActivation.

Generalizations

- ActivityNodeActivation

Attributes

None

Associations

None

Operations

[1] getMatchingExceptionHandlers (in exception : Value) : ExceptionHandler [0..*]

```
// Return the set of exception handlers that have an exception type
// for which the given exception is an instance.

ExceptionHandlerList handlers = ((ExecutableNode)this.node).handler;
ExceptionHandlerList matchingHandlers = new ExceptionHandlerList();

for (int i = 0; i < handlers.size(); i++) {
    ExceptionHandler handler = handlers.getValue(i);

    boolean noMatch = true;
    int j = 1;
    while (noMatch & j <= handler.exceptionType.size()) {
        if (exception.isInstanceOf(handler.exceptionType.getValue(j - 1))) {
            matchingHandlers.addValue(handler);
            noMatch = false;
        }
        j = j + 1;
    }
}

return matchingHandlers;
```

[2] handle (in exception : Value, in handler : ExceptionHandler)

```
// Offer the given exception to the body of the given exception handler
// on its exception input node.

ActivityNodeActivation handlerBodyActivation =
    this.group.getNodeActivation(handler.handlerBody);
ActivityNodeActivation inputActivation =
    handlerBodyActivation.group.getNodeActivation(handler.exceptionInput);

ObjectToken token = new ObjectToken();
token.value = exception;
inputActivation.addToken(token);

handlerBodyActivation.receiveOffer();
```

[3] propagateException (in exception : Value)

```
// If there is no matching exception handler for the given exception, then propagate
// the exception to either the containing node activation or the activity execution, as
// appropriate.
// If there is a matching exception handler, then use it to catch the exception.
// (If there is more than one matching handler, then choose one non-deterministically.)

ExceptionHandlerList matchingExceptionHandlers =
        this.getMatchingExceptionHandlers(exception);

if (matchingExceptionHandlers.size() == 0) {
    this.terminate();
    if (this.group.containingNodeActivation != null) {
        this.group.containingNodeActivation.propagateException(exception);
    } else {
        this.group.activityExecution.propagateException(exception);
    }
} else {
    ChoiceStrategy strategy = (ChoiceStrategy) this.getExecutionLocus().
            factory.getStrategy("choice");
    ExceptionHandler handler = matchingExceptionHandlers.getValue(
            strategy.choose(matchingExceptionHandlers.size()) - 1);
    this.handle(exception, handler);
}
```

8.9.2.14 FlowFinalNodeActivation

A flow final node activation is a control node activation for a node that is a flow final node.

Generalizations

- ControlNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
//Consume all incoming tokens.
for (int i = 0; i < incomingTokens.size(); i++) {
     Token token = incomingTokens.getValue(i);
     token.withdraw();
}
```

8.9.2.15 ForkedToken

A forked token is a proxy for a token that has been offered through a fork node. If the token is accepted through the fork node, then the original token is withdrawn from its holder, but the forked token remains held by the fork node activation until all outstanding offers on all outgoing edges are accepted.

Generalizations

- Token

Attributes

- baseTokenIsWithdrawn : Boolean
  Indicates whether withdraw() has been classed on the base token.
- remainingOffersCount : Integer
  The remaining number of outstanding offers for this token on outgoing edges of the fork node.

Associations

- baseToken : Token

Operations

[1] copy ( ) : Token

```
// Return a copy of the base token.

return  this.baseToken.copy();
```

[2] equals (in otherToken : Token) : Boolean

```
// Test if this token is equal to another token.

return this == otherToken;
```

[3] getValue ( ) : Value [0..1]

```
// Return the value of the base token.

return  this.baseToken.getValue();
```

[4] isControl ( ) : Boolean

```
// Test if the base token is a control token.

return  this.baseToken.isControl();
```

[5] withdraw ( )

```
// If the base token is not withdrawn, then withdraw it.
// Decrement the remaining offers count.
// When the remaining number of offers is zero, then remove this token from its holder.

if  (!this.baseTokenIsWithdrawn  &  !this.baseToken.isWithdrawn())  {
     this.baseToken.withdraw();

     // NOTE: This keeps a base token that is a forked token from being
     // withdrawn more than once, since withdrawing a forked token may
     // not actually remove it from its fork node holder.
     this.baseTokenIsWithdrawn  =  true;
}
if  (this.remainingOffersCount  >  0)  {
     this.remainingOffersCount  =  this.remainingOffersCount  -  1;
}
```

```
if (this.remainingOffersCount == 0) {
     super.withdraw();
}
```

8.9.2.16 ForkNodeActivation

A fork node activation is a control node activation for a node that is a fork node.

Generalizations

- ControlNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// Create forked tokens for all incoming tokens and offer them on all outgoing edges.

ActivityEdgeInstanceList outgoingEdges = this.outgoingEdges;
int outgoingEdgeCount = outgoingEdges.size();

TokenList forkedTokens = new TokenList();
for (int i = 0; i < incomingTokens.size(); i++) {
     Token token = incomingTokens.getValue(i);
     ForkedToken forkedToken = new ForkedToken();
     forkedToken.baseToken = token;
     forkedToken.remainingOffersCount = outgoingEdgeCount;
     forkedToken.baseTokenIsWithdrawn = false;
     forkedTokens.addValue(forkedToken);
}

this.addTokens(forkedTokens);

this.sendOffers(forkedTokens);
```

[2] terminate ( )

```
// Remove any offered tokens and terminate.
```

```
super.terminate();
this.clearTokens();
```

8.9.2.17 InitialNodeActivation

An initial node activation is a control node activation for a node that is an initial node.

Generalizations

- ControlNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// Create a single token and send offers for it.

TokenList tokens = new TokenList();
tokens.addValue(new  ControlToken());
this.addTokens(tokens);

this.sendOffers(tokens);
```

8.9.2.18 JoinNodeActivation

A join node activation is a control node activation for a node that is a join node.

Generalizations

- ControlNodeActivation

Attributes

None

Associations

None

Operations

[1] isReady ( ) : Boolean

```
// Check that all incoming edges have sources that are offering tokens.
```

```
boolean ready = true;
int i = 1;
while (ready & i <=this.incomingEdges.size()) {
     ready = this.incomingEdges.getValue(i-1).hasOffer();
    i = i + 1;
}

return ready;
```

8.9.2.19 MergeNodeActivation

A merge node activation is a control node activation for a node that is a merge node.

Generalizations

- ControlNodeActivation

Attributes

None

Associations

None

Operations

None

8.9.2.20 ObjectNodeActivation

An object node activation is an activity node activation for a node that is an object node.

Generalizations

- ActivityNodeActivation

Attributes

- offeredTokenCount : Integer
  The number of held tokens that have already been offered.

Associations

None

Operations

[1] addToken (in token : Token)

```
// Transfer the given token to be held by this node only if it is a non-null object token.
// If it is a control token or a null token, consume it without holding it.
```

```
if (token.getValue() == null) {
     token.withdraw();
} else {
     super.addToken(token);
}
```

[2] clearTokens ( )

```
// Remove all held tokens.

super.clearTokens();
this.offeredTokenCount = 0;
```

[3] countOfferedValues ( ) : Integer

```
// Count the total number of non-null object tokens being offered to this node activation.

int totalValueCount = 0;
int i = 1;
while (i <= this.incomingEdges.size()) {
     totalValueCount = totalValueCount + this.incomingEdges.getValue(i-
1).countOfferedValues();
    i = i + 1;
}

return totalValueCount;
```

[4] countUnofferedTokens ( ) : Integer

```
// Return the number of unoffered tokens that are to be offered next.
// (By default, this is all unoffered tokens.)

if (this.heldTokens.size() == 0) {
     this.offeredTokenCount = 0;
}

return this.heldTokens.size() - this.offeredTokenCount;
```

[5] getUnofferedTokens ( ) : Token [0..*]

```
// Get the next set of unoffered tokens to be offered and return it.
// [Note: This effectively treats all object flows as if they have weight=*, rather than the
weight=1 default in the current superstructure semantics.]
```

```
TokenList tokens = new TokenList();

int i = 1;
while (i <= this.countUnofferedTokens()) {
     tokens.addValue(this.heldTokens.getValue(this.offeredTokenCount + i - 1));
    i = i + 1;
}

return tokens;
```

[6] removeToken (in token : Token) : Integer

```
// Remove the given token, if it is held by this node activation.

int i = super.removeToken(token);
if (i > 0 & i <= this.offeredTokenCount) {
     this.offeredTokenCount = this.offeredTokenCount - 1;
}

return i;
```

[7] run ( )

```
// Initialize the offered token count to zero.

super.run();
this.offeredTokenCount = 0;
```

[8] sendOffers (in tokens : Token [0..*])

```
// If the set of tokens to be sent is empty, then offer a null token instead.
// Otherwise, offer the given tokens as usual.

if (tokens.size() == 0) {
     if (tokens.size() == 0) {
          ObjectToken token = new ObjectToken();
          token.holder = this;
          tokens.addValue(token);
     }
}

super.sendOffers(tokens);
```

[9] sendUnofferedTokens ( )

```
// Send offers over all outgoing edges, if there are any tokens to be offered.

TokenList tokens = this.getUnofferedTokens();
this.offeredTokenCount = this.offeredTokenCount + tokens.size();

this.sendOffers(tokens);
```

[10] takeUnofferedTokens ( ) : Token [0..*]

```
// Take the next set of unoffered tokens to be offered from this node activation and return
them.

TokenList tokens = this.getUnofferedTokens();
for (int i = 0; i < tokens.size(); i++) {
     Token token = tokens.getValue(i);
     token.withdraw();
}
return tokens;
```

[11] terminate ( )

```
// Terminate the node activation and remove any held tokens.

super.terminate();
this.clearTokens();
```

8.9.2.21 ObjectToken

An object token represents the passing of data along an object flow edge.

Generalizations

- Token

Attributes

None

Associations

- value : Value [0..1]
  The value carried by this token. A token may have no value, in which case it is a “null token.”

Operations

[1] copy ( ) : Token

```
// Return a new object token with the same value as this token.
// [Note: the holder of the copy is not set.]

ObjectToken copy = new ObjectToken();
copy.value = this.value;

return copy;
```

[2] equals (in other : Token) : Boolean

```
// Test if this object token is the same as the other token.

return this == other;
```

[3] getValue ( ) : Value [0..1]

```
// Return the value of this object token.

return this.value;
```

[4] isControl ( ) : Boolean

```
// Return false for an object token.

return false;
```

8.9.2.22 Offer

An offer is a group of tokens offered together. The grouping of offered tokens into offers usually does not matter for how the tokens may be accepted. However, control and object tokens may become grouped together in the same offer due to a join node that has both incoming control and object flows. In this case, the control tokens are implicitly accepted once all the object tokens in the same offer have been accepted.

Generalizations

None

Attributes

None

Associations

- offeredTokens : Token [0..*]

Operations

[1] countOfferedValues ( ) : Integer

```
// Return the number of values being offered on object tokens.
// Remove any tokens that have already been withdrawn and don't include them in the count.

this.removeWithdrawnTokens();
```

```
int count = 0;
for (int i = 0; i < this.offeredTokens.size(); i++) {
     if (this.offeredTokens.getValue(i).getValue() != null) {
         count = count + 1;
    }
}

return count;
```

[2] getOfferedTokens ( ) : Token [0..*]

```
// Get the offered tokens, removing any that have been withdrawn.

this.removeWithdrawnTokens();

TokenList tokens = new TokenList();
TokenList offeredTokens = this.offeredTokens;
for (int i = 0; i < this.offeredTokens.size() ; i++) {
     Token offeredToken = offeredTokens.getValue(i);
     tokens.addValue(offeredToken);
}

return tokens;
```

[3] hasTokens ( ) : Boolean

```
// Check whether this offer has any tokens that have not been withdrawn.

this.removeWithdrawnTokens();
return this.offeredTokens.size() > 0;
```

[4] removeOfferedValues (in count : Integer)

```
// Remove the given number of non-null object tokens from those in this offer.

int n = count;
int i = 1;
while (n > 0) {
     if (this.offeredTokens.getValue(i-1).getValue() != null) {
          this.offeredTokens.removeValue(i-1);
    } else {
         i = i + 1;
    }
    n = n - 1;
}
```

[5] removeWithdrawnTokens ( )

```
// Remove any tokens that have already been consumed.

TokenList offeredTokens = this.offeredTokens;
int i = 1;
while (i <= this.offeredTokens.size()) {
     if (this.offeredTokens.getValue(i-1).isWithdrawn()) {
         this.offeredTokens.remove(i-1);
         i = i - 1;
    }
    i = i + 1;
}
```

8.9.2.23 Token

A token is an individual element of data or control that may flow across an activity edge.

Generalizations

None

Attributes

None

Associations

- holder : ActivityNodeActivation [0..1]

Operations

[1] copy ( ) : Token

```
// Make a copy of this token.
```

[2] equals (in other : Token) : Boolean

```
Test if this token is equal to another token.
```

[3] getValue ( ) : Value [0..1]

```
Get the value associated with this token (if any).
```

[4] isControl ( ) : Boolean

```
Test whether this is a control token.
```

[5] isWithdrawn ( ) : Boolean

```
// Test if this token has been withdrawn.
```

```
return this.holder == null;
```

[6] transfer (in holder : ActivityNodeActivation) : Token

```
// If this token does not have any holder, make the given holder its holder.
// Otherwise, remove this token from its holder and return a copy of it transferred to a new holder.

Token token = this;
if (this.holder != null) {
     this.withdraw();
     token = this.copy();
}

token.holder = holder;
return token;
```

[7] withdraw ( )

```
// Remove this token from its holder, withdrawing any offers for it.

if (!this.isWithdrawn()) {
     this.holder.removeToken(this);
     this.holder = null;
}
```

8.10 Actions

8.10.1 Overview

This subclause describes the semantics of actions, which are the basic units out of which most non-trivial kinds of behavior are created in fUML. Actions are kinds of activity nodes, so they are always executed in the context of an activity, which is the overall behavioral construct (see 8.9 for the general semantics of activities). Subclause 7.11 gives the abstract syntax for actions included in fUML. The present subclause defines the fundamental semantics for actions and pins, as well as the specific semantics of those kinds of actions included in the fUML subset.

Action Activation

Since actions are kinds of activity node, the semantic visitor classes for actions are kinds of activity node activations (see 8.9.1). In addition, the pins on actions are also kinds of activity nodes (specifically, object nodes) so there are also visitor classes for input and output pins that are kinds of object node activation. Pin activations are associated with an action activation in a parallel way to the association of the corresponding pins with their action (see Figure 8.27).

The ActionActivation class provides a method for the abstract fire operation inherited from ActivityNodeActivation and overrides the takeOfferedTokens operation. In general, the fire operation for an activity node activation is called whenever the prerequisites for execution of the node have been satisfied, as determined by the isReady operation, after accepted tokens are obtained using takeOfferedTokens (see 8.9.1). For an action activation, these operations are specialized to model the particular semantic requirements of action execution in terms of the offers received by the pins of the action.

When an input pin activation receives an offer, via a call to its receiveOffer operation, it passes the offer on by calling the receiveOffer operation of its action activation. The isReady operation of the action activation then checks whether all its input pin activations are ready. If so, then it fires all of the input pin activations at once, accepting all the offers that have been made to them and moving the accepted tokens to the input pins.

Note: The UML 2 Specification (subclause 16.2.3.4) states that "Executing an Action in an Activity requires all of its InputPins to be offered all necessary tokens, as specified by their minimum multiplicity (except for the special cases of ActionInputPins and ValuePins, as discussed above). When the Action begins executing, all the InputPins accept tokens offered to them at once, up to the maximum multiplicity allowed on each InputPin.." In the execution model, the calls to the isReady and takeOfferedTokens operations from the ActivityNodeActivation::receiveOffer operation are made within an isolated region (see 8.9.1). This means that the source node activations of any offers to an action activation cannot be modified while the action activation is checking for, and possibly accepting, offers to its input pins. This prevents contention for the offers during this period, as required by the UML 2 semantics. (See below for more on the semantics of isolated regions – that is, structured activity nodes with mustIsolate = true.)

The above behavior is specified in the takeOfferedTokens method for ActionActivation and is generic to all action activations. The actual specific behavior of each kind of action is factored into the doAction operation. This operation is called from the ActionActivation fire operation after all the input pin activations have fired. Once the doAction operation is complete, all the output pin activations of the action are fired, which causes them to send offers on any outgoing edges (assuming they have tokens to offer), and a control token is offered on all control flows outgoing from the action.

The semantics of the offering of a token on control flows outgoing from an action are those of an "implicit fork" (see subclause 15.5.3.1 of the UML 2 Specification). Therefore, in order that the semantic model for control flows from an action be identical to those of a fork, if an action has outgoing control flows, an anonymous fork node activation is also created along with the action activation for the action. The action activation is then connected to the fork node activation, and the activity edge instances corresponding to the control flows outgoing the action are connected to the fork node activation. When the action activation completes its behavior, it creates a control token and offers it to the fork node activation which, per the semantics of fork nodes, in turn offers it on all outgoing activity edge instances.

Invocation Actions

The basic invocation actions include send signal, call operation, and call behavior actions (see 7.11). The corresponding activation classes are specializations of ActionActivation (see Figure 8.28).

Of these, the behavior of a send signal action activation is the simplest. When it fires, it takes values from its argument input pin activations, constructs a signal instance with slots filled in with those values, creates a signal event occurrence referencing the signal instance and sends the event occurrence via the object reference obtained from its target input pin activation. The event occurrence is sent by calling the send operation on the target object (see 8.8.1). This results in the event occurrence being placed in the event pool for the target object, at which point the send call returns and the thread on which the send signal action is executing can continue (as appropriate). As discussed in 8.8.1, event occurrences in the event pool are dispatched asynchronously by the event dispatching loop of the object activation for the target object.

In contrast to sending signals, call behavior and operation actions in fUML are always synchronous (see 7.11). This basic synchronous calling behavior is modeled in the doAction method of the CallActionActivation class. Associated with this class is an Execution object that represents the invocation of the called behavior. Other than for how this execution object is instantiated, the semantics of call behavior and call operation actions are the same:

1. Values of argument input pin activations are passed as input parameter values to the execution object.
2. Listeners are registered for the output pin activations corresponding to streaming output parameters.
3. The execute operation is called on the execution.

4. If there are no streaming input parameters, then the invocation is complete once the execute operation returns. Otherwise, the invocation continues until the invoked execution terminates (and, since the call action must be locally non-reentrant in this case, the action cannot fire again until the invocation is complete).
5. Once the invocation is complete, if the execution completes without an exception, any non-streaming output parameter values are placed on result output pin activations. If the execution raises an exception, then this is propagated using the general propagateException operation inherited from ActionActivation. In either case, the execution object is destroyed.

Note: The fUML execution model interprets the semantics of called behaviors as requiring that the execution object instantiated by a synchronous call action be destroyed when the call returns. Otherwise, repeated calls would result in a potentially large number of anonymous, completed, called executions accumulating at any execution locus.

Instantiating the execution object for a call behavior action is straightforward: an instance of the referenced behavior is simply instantiated at the execution locus of the call action. Instantiating the execution object for an operation call, on the other hand, requires that a potentially polymorphic operation be dispatched in order to determine which method should act as its behavior. This dispatching is carried by calling the dispatch operation on the target object. (See 8.7.1 for further discussion of polymorphic operation dispatching.)

Unlike the case of a call behavior action, the default behavior of polymorphic operation dispatching is for the execution object for the operation method to be instantiated at the locus of the target object, not the locus of the action execution. Thus, if an operation call is made on an object at a remote locus, then the operation will be executed on that locus. While fUML does not provide a normative means for passing object references between loci, a specific execution tool may implement such a means, in which case the semantics of operation calls across inter-locus references is specified normatively. (See 8.3.1 for a discussion of loci.)

Note: As described in 8.7.1, polymorphic operation dispatching is a semantic variation point. The default semantics for operation calls acts as described above for references to objects on other loci, and it would generally be expected that any alternative dispatching strategy would have a similar behavior. However, a conforming execution tool is allowed to define a dispatching strategy that would prescribe, for example, that all operation executions are performed on the local locus, regardless of where the target object resides.

Start Object Behavior Action

A start object behavior action is used to start the execution of an instantiated behavior or a classifier behavior of an active object.

In general UML, a start object behavior action is a kind of call action and its behavior invocation may be either synchronous or asynchronous. However, in fUML, only asynchronous invocation is supported (see 7.11). Therefore, StartObjectBehaviorActionActivation is a subclass of InvocationActionActivation rather than CallActionActivation (see Figure 8.28), since CallActionActivation only provides synchronous call semantics.

If the input object to the action activation is an active object of the type of the input pin of the start object behavior action, then the effect of the action is to start the classifier behavior associated with that type. The start object behavior activation doAction method calls the startBehavior operation on the input object, passing as parameter values the values on the argument input pin activations for the action activation. This results in the input object having an object activation with a classifier behavior execution for that classifier behavior (see 8.8). Once the classifier behavior execution starts, it executes concurrently, so the execution thread of the start object behavior action activation can continue without blocking. If the classifier behavior for the indicated type already has an execution for the given object, then the action has no effect.

If the input pin to the start object behavior action activation does not have a type, then the effect of the action is to start the classifier behaviors for all types of the input object that have classifier behaviors that are not already executing. Note that, in this case, it is not possible to specify input parameter values for the classifier behaviors.

If the input object to the action activation is itself an execution object (i.e., an instance of a behavior), then the effect is to start the execution of the behavior (if it isn't already executing). However, since the execution most proceed asynchronously to the execution of the start object behavior action, it is necessary to start a new execution thread. This is achieved in the execution model by starting the execution of the behavior in the same way as a classifier behavior. That is, the input execution object is given an object activation with a classifier behavior execution that provides a new execution thread for the execution object itself. The execute operation for the input execution object is called on this new thread, so that the thread of the start object behavior action activation can continue without blocking. (See also 8.8.)

Note: fUML also includes the start classifier behavior action. This acts similarly to a start object behavior action, but it only handles active objects with classifier behaviors and it does not provide a mechanism for specifying input parameters. Start classifier behavior actions are supported in fUML for compatibility with past practice, but they should be considered deprecated in favor of start object behavior actions.

Object Actions

fUML includes the following object actions (see Figure 8.29 for the corresponding activation model).

- Create Object Action – In fUML, the classifier specified by a create object action must be a class (see 7.11). Therefore, the instance created really is an object, as the action name indicates. The object becomes part of the extent of the specified class at the execution locus of the activity execution that contains the action activation. (See 8.3 and 8.7 for more on loci and extents.) The classifier for a create object action may be a behavior, in which case the instance created is an unstarted behavior execution that may be started asynchronously using a start object behavior action (see above). However, in fUML, the classifier may *not* be an owned behavior (see 7.11), because starting an instance of such a behavior asynchronously would not allow for the execution context (which must be an instance of its context classifier) to be set properly. An owned behavior may be called synchronously or, if it is a classifier behavior, it may be started asynchronously using a start object behavior action on an instance of its (owning) context classifier (see above).
- Destroy Object Action – This action accepts an object reference and destroys the referenced object. Destruction involves terminating the object activation (if any), removing all of the objects types, and removing the object from the extent at the execution locus. Note that the fUML semantics do not preclude references continuing to exist to destroyed objects. However, since such objects do not have any types, they will have neither attributes nor behaviors. Note also that only objects can be destroyed in fUML-attempting to destroy a data value will have no effect.
- Test Identity Action – This action tests whether its two input values are “identical.” If the input values are both data values, then “identical” means that they are either the same primitive value or they have the same compound type, with identical values for all corresponding attributes. If the input values are both object references, then “identical” means that they reference the same object. That is, for data values equality is testing “by value,” while for objects it is tested “by reference.”
- Read Self Action – This action reads the context object of the activity activation that contains the action activation. If the activity is associated with a class (as a method or a classifier behavior), then the context object will be an instance of that class. Otherwise, the context object will be the execution object of the activity itself.
- Value Specification Action – This action evaluates a given value specification and outputs the resulting value. The value specification is evaluated using the evaluate operation of the executor at the execution locus of the activity execution that contains the action activation (see 8.3).

Object Classification Actions

fUML also includes three additional actions related to object classification (see Figure 8.29 for the corresponding activation model).

- Read Extent Action – This action is used to obtain the extent of a class at the execution locus of the activity activation containing the action activation. The action outputs references to each of the objects in the extent. Note that the extent of a class is considered to also include the instances of all subclasses of the identified class. In fUML, the classifier associated with this action must be a class (see 7.11).

- Read Is Classified Object Action – This action is used to test whether its input value is of a given type. The input value does not actually have to be an object reference but can also be a data value. The action produces a true output if the input has the given classifier as one of its types (objects may have multiple types). If isDirect is false, then the action also outputs true of any type of the input value is a specialization (directly or indirectly) of the given classifier. Otherwise, the action outputs false.
- Reclassify Object Action – This action is used to change the type(s) of an object. In fUML, the input value must be a reference to an object and all the classifiers associated with the action must be classes (see 7.11). Per the semantics of UML 2 for reclassification (see subclause 16.4.3.7 of the UML 2 Specification), this action removes the indicated old classifiers as types of the input object and adds the new classifiers, taking into account cases in which a classifier my be in both sets. Note that, if a classifier with a classifier behavior is removed, then any execution the input object may have for this behavior is terminated (see also 8.8). However, if a classifier is added with a classifier behavior, this behavior is not started until an explicit start object behavior action is performed (see above).

Link Actions

In fUML, a link is an extensional value that exists at a specific execution locus (see 8.7). Unlike objects, however, there are no explicit references to links. Rather, links may be thought of as tuples of values, one for each association end, and a link of a specific association can be identified by giving such a tuple. Note that this identification is not necessarily unique, though, unless all the ends of the association are specified as being unique.

fUML includes the following actions for manipulating links (see Figure 8.30 for the corresponding activation model).

- Create Link Action – Given a value for each association end, this action normally creates a link with those values. This link becomes a member of the extent of the association at the execution locus of the activity activation that contains the action activation. However, if a link already exists in the association extent with the same tuple of values, and all the ends of the association are specified as unique, then no new link is actually created (though this is not an error). Since, in fUML, an association always owns its ends (see 7.7), each of the values for the link are represented as structural feature values for the link (see 8.7 for more on the representation of the structure of links). If an association end is ordered, then the link also maintains the position of its value for that association end relative to the value provided by other links in the extent of the association.
- Destroy Link Action – Given a value for each association end, this action destroys all links that match the link end destruction data in the extent of the given association at the execution locus of the activity activation that contains the action activation. Destroying a link means simply removing it from the extent of the association. Matching links are determined as follows:
    - For unique ends, or non-unique ends for which isDestroyDuplicates is true, match links with a matching value for that end.
    - For non-unique, ordered ends for which isDestroyDuplicates is false, match links with an end value at the given destroyAt position.
    - For non-unique, non-ordered ends for which isDestroyDuplicates is false, pick one matching link (if any) nondeterministically.

  Note: The behavior in this third class when there is more that one matching link is not explicitly stated in the UML 2 Specification (subclause \16.6.3.4). fUML provides an interpretation of nondeterministic choice in this case.
- Read Link Action – This action provides a means for querying the extent of an association at the execution locus of the activity activation that contains the action activation. The action specifies values for all ends of the association but one-

the open end (see 7.11). This link end data identifies a subset of matching links from the association extent that have the specified end values. The action outputs the set of values on the open ends of these matching links.

- Clear Association Action – This action destroys all the links in the extent of the given association (at the execution locus of the activity activation that contains the action activation) that has the input value of the action as an end value.

Structural Feature Actions

fUML includes actions for accessing the structural features of both objects and data types (see Figure 8.31 for the corresponding activation model).

Note: The UML Specification (subclause 16.8.3.3) states that, for an add structural feature value action, "The semantics is undefined for adding a value that violates the upper multiplicity of the StructuralFeature…." Nevertheless, the fUML semantics *are* defined in this case, such that the given value is always added, even though it violates the upper multiplicity of the structural feature.

Note: For an ordered structural feature, the UML Specification (subclause 16.8.3.3) defines the effect of "insertAt" to be: "An insertion point that is a positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values, with the integer one meaning the new value will be first in the sequence. A value of unlimited ("*") for the insertion point means to insert the new value at the end of the sequence." For fUML, this behavior is assumed to mean that the new value is inserted into the required position without replacing any of the previously existing values in the structural feature, which retain the same relative ordering as before the insertion of the new value.

Unmarshall Action

An unmarshall action takes a structured value as input and returns the values for each of the attributes of the input on corresponding result output pins. The number of result pins must correspond statically to the number of attributes of the type of the input pin (including inherited attributes). See Figure 8.32 for the corresponding activation model.

Accept Event Action

An accept event action is used in an activity to wait for the occurrence of a specific event. In fUML, an accept event action is either a regular accept event action, in which case it can only wait for signal events, or it is a specialized accept call action, in which case it can only wait for call events (see 7.11). The discussion on accept event actions below also applies to accept call actions. This is followed by some additional discussion specific to accept call actions.

To wait for the dispatching of a signal event, the accept event action activation must register itself as an event accepter with its context object. Actually, the action activation does not directly register itself, but, instead, it creates an accept event action event accepter object, which is a kind of event accepter (see 8.8), and registers this with the context object (see Figure 8.32). This registration happens when the accept event action activation fires (the only prerequisites for this action are that it receives a control token), and the doAction operation is not called in this case.

Instead, the behavior of the action is triggered when an event occurrence is dispatched from the event pool of the context object that matches the trigger specification of the accept event action. In this case, the object activation dispatchEvent operation calls the accept operation on the accept event action event accepter object (see 8.8). The event accepter then forwards the call to the accept operation of the accept event action activation, starting a new thread within the activity containing the start object behavior action.

Accept Call and Reply Actions

An accept call action is a specialized accept event action used to wait for call events. It registers and accepts event occurrences as described for accept event actions in general above. However, when triggered, in addition to producing the unmarshalled values of the input parameters (if any) of the called operation, an accept call action produces another output, the return information for the call. Return information is a special value that may be passed on an object flow of an activity, but is only usable as input to a reply action, in order to return from a call. The return information contains a link back to the call event occurrence that triggered the accept call action for the call.

When a reply action fires, it takes values for the output parameters (if any) of the called operation from its replyValue input pins and a return information value from its returnInformation pin. It then returns the output parameter values to the caller using the reply operation on the return information. This reply operation sets the output parameter values via the call event occurrence and releases the calling thread, which will have been suspended waiting for a reply to the call (see 8.8).

Note: A reply action is associated with a trigger that identifies the call event from which the reply action is returning. The UML Specification (subclause 11.3.43) states that "The semantics are also undefined if the return information value is not for a call to the same Operation as identified by the replyToCall Trigger of the ReplyAction." In fUML, this is interpreted as meaning that the operation specified by the call event on the trigger must be the same as the operation that was called by the call event occurrence on the return information provided to the call action. If the operations do not match, the reply action has no effect.

Structured Activity Node Activation

Unlike other kinds of actions, structured activity nodes have nested activity nodes within them. As shown in Figure 8.33, the activation of the nested activity nodes is handled by an activity node activation group associated with the structured activity node activation (see 8.9 for the specification of ActivityNodeActivationGroup).

Note that all structured activity node activations have exactly one activation group that covers the activation of all nested activity nodes. However, how nested activity nodes are actually activated varies depending on the kind of structured activity node.

For the base structured activity node, which simply groups its nested activity nodes, execution proceeds much as in the case of an overall activity. All nested activity nodes are activated, and subsequent behavior is determined by the flow of offers and tokens between activations.

For a conditional node, however, the test part is activated first. Depending on the result of the test, additional nodes are activated depending on which conditional clause is selected.

For a loop node, the loop test and body parts are repeatedly activated (with the test coming before or after the body, depending on the isTestedFirst attribute of the loop node). The same activity node activation group is used for every iteration of the loop, but the group is cleared of node activations between iterations.

Isolation

If a structured activity node has the property mustIsolate = true, then its activity node activations run in isolation from activity node activation external to it. The UML Specification (subclause 16.11.3.2) defines this behavior as follows:

> If the mustIsolate flag is true for a StructuredActivityNode, then any access to an object by an Action within the node must not conflict with access to the object by an Action outside the node. A conflict is defined as an attempt to write to the object by one or both of the Actions. If such a conflict potentially exists, then no such access by an Action outside the isolated StructuredActivityNode may be interleaved with the execution of the StructuredActivityNode.

For the purposes of fUML, however, it is important to define this important optional behavior somewhat more completely. The following definitions apply for the purposes of this discussion.

- An execution trace provides timing for all the events in the execution of a model.

- The duration of a firing of an action activation is the time interval from the event of the action activation firing to the event of the action activation offering tokens on outgoing control flows (even if there are no outgoing control flows, the duration ends at that point in time at which the firing of the action activation is “complete” and would offer control tokens if there were flows). A legal execution trace is one that is permitted by the behavioral semantics specified for executing the model. Note that there can, and generally will, be multiple possible legal execution traces for any given model.
- Two action activation firings overlap if their durations are not disjoint.
- An action activation A is serializable with respect to another action activation B if, for any legal execution trace in which one or more of the firings of A and B overlap, there is another legal execution trace in which none of their firings overlap but for which the execution behavior of the firings of B are identical to that of the first trace. (Note that the behavior of A does not have to be preserved in the second trace. This means that A being serializable with respect to B does not necessarily imply that B is serializable with respect to A.)
- The scope of control of an activity execution or a structured activity node activation firing is defined to be the set of activity node activations covered by the following:
  - For a structured activity node activation, that activation itself.
  - All activations of nested activity nodes with the activity or structured activity node that are run as a result of that specific activity execution or structured activity node activation firing. (In the execution model, this is called the “activity node activation group”.)
  - The scope of control of the firing of any nested structured activity node activations.
  - The scope of control of any activity executions resulting from the firing of any nested call (behavior or operation) actions (which, in fUML, are always synchronous).

The rule for isolation can now be stated fairly simply: Let S be a structured activity node with mustIsolate=true. Then any action activation not in the scope of control of S must be serializable with respect to any action activation that is within the scope of control of S.

Basically, under this rule, any action behavior not under the control of S, even if it physically happens in parallel with an execution of S, has the same effect on S as if it occurred entirely before or entirely after the execution of S. In particular, any actions that write to objects read within S must either have their effect visible throughout the execution of S (“as if it occurred entirely before the execution of S”) or their effect must not be visible at all within the execution of S (“as if it occurred entirely after the execution of S”). (This is similar to the way that “isolation” is defined for database transactional semantics.)

Note that the asymmetric definition of “serializable” above means that, in general, an action activation not under the control of S can see into intermediate results produced by S (in database terminology, this is known as a “dirty read”), unless it, too, is part of some other structured activity node with mustIsolate=true. For two structured activity nodes to run in complete isolation with respect to each other, both must specify mustIsolate=true.

Note also that the above rule does not allow certain deadlock conditions that can occur due to specific implementation techniques, such as locking. For example, there is the archetypical case in which two concurrent threads are each holding locks which the other needs, and so neither can proceed. However, in most such cases, there is a legal execution trace in which these threads could have successfully executed (e.g., if they were run sequentially instead of concurrently). The intent is that the execution trace leading to deadlock would not be legal at all, since it is only the locking implementation that leads to the deadlock, not anything specified by the behavioral semantics. In particular, this means that, if an execution tool uses locking to implement isolation, then it also must provide some means to detect implementation-specific deadlock conditions and recover from them (again, this is typically what is done in database transaction implementations).

On the other hand, there are cases in which deadlock cannot be avoided. For example, suppose a structured activity node with mustIsolate = true contains just two read actions. The first read action has an outgoing control flow that crosses out of the structured activity node to a write action on the outside that writes to the object read by the read actions. If the write action then has an outgoing control flow that crosses back into the structured activity node to the second read action, it is impossible to satisfy both the control flow constraints and the isolation rule. Such a model has no legal control flows. Per the UML Superstructure Specification, it is actually ill-formed and has no execution semantics.

Note: The above semantics for mustIsolate = true are intended to allow the simple implementation of approach of serializing the execution of all structured activity nodes with mustIsolate = true-that is, running them sequentially, one at a time, with nothing else running at the same time. One subtlety here is the case when an execution of one or more of the isolated structured activity nodes does not terminate, due to, say, an infinite loop. In this case, there may not be any finite execution trace in which all isolated structured activity nodes can complete sequentially. However, since there are no particular requirements in the fUML semantics for liveliness or fairness in concurrent execution, it is generally permissible in any case for an implementation to allow a concurrent thread that does not terminate to continue to use all resources and not allow any other threads to run. Therefore, the rule above for isolation is not meant to disallow a fully serialized implementation.

The above rule for isolation is part of the base semantics of the modeling subset used to write the execution model itself (see 10.4.5). Therefore, structured activity nodes with mustIsolate = true may be used within the execution model. For fUML user models being executed by the execution model, the effect of mustIsolate = true is achieved by activating the body of the fUML structured activity node within a structured activity node in the execution mode with mustIsolate = true. This results in the body of the structured activity node being run in isolation from other threads running within the executing fUML activity, resulting in the base isolation behavior being elevated to fUML.

In order to accommodate this optional isolation behavior, the class StructuredActivityNodeActivation provides a method for the operation doAction in terms of an operation called doStructuredActivity. The operation StructuredActivityActivation:: doAction checks the mustIsolate flag for the structured activity node being executed. If it is true, then doAction calls doStructuredActivity within a structuredActivityNode with mustIsolate = true. If mustIsolate = false, then doAction still calls doStructuredActivity, but not within an isolated structured activity node.

The classes ConditionalNodeActivation and LoopNodeActivation specialize StructuredActivityNodeAcivation (see Figure 8.33). They both override the operation doStructuredActivity to specify the behavior specific to conditional nodes and loop nodes. However, they do not override the doAction operation, and, therefore, they inherit the basic isolation behavior from StructuredActivityNode behavior.

Collections

The UML 2 Specification (subclause 16.12.3) defines the semantics of an expansion region as “a StructuredActivityNode that takes as input one or more collections of values and executes its contained ActivityNodes and ActivityEdges on each value in those collections” where:

> An execution engine may define various kinds of collection types that it supports (sets, bags, and so on), individual instances of which may be constructed from element values and from which those element values may later be obtained. Such a collection instance is passed as a single value on a single token. An execution engine may alternatively support collections implicitly as the set of values passed in a group of tokens placed together on an ExpansionNode.

Neither the fUML subset nor the Foundation Model Library provide a standard set of collection types. Instead, fUML relies on the use of properties with multiplicity upper bounds greater than zero to provide the ability to model collections.

Therefore, rather than an expansion node being expected to receive a single token with a collection value, in fUML the “collection” is made up of the values on a set of tokens accepted by the expansion node. An expansion region fires when its input expansion node accepts an offer for such a collection of tokens. If the region has more than one input expansion region, then all must accept the same number of tokens for the region to fire.

Similarly, the output expansion nodes of the region (if any) collect tokens generated during the iterations of the body of the region. When the expansion region completes, the tokens on its output expansion nodes are offered downstream in the normal fashion.

Expansion Region Activation

An expansion region is a kind of structured activity node and, therefore, a kind of action. However, because the semantics of expansion regions are rather different than those of other structured activity nodes, ExpansionRegionActivation does not specialize StructuredActivityNodeActivation but, rather, directly specializes ActionActivation (see Figure 8.34). There is also an ExpansionNodeActivation class to capture the specialized semantics of expansion nodes.

Unlike other structured activity nodes (as described above), an expansion region activation may have multiple activity node activation groups. This is to allow for the possible parallel activation of the body of the expansion region, if so specified for the expansion region. In addition, the activity node activation groups for an expansion region activation are all instances of the specialized ExpansionActivationGroup. This specialization handles the semantic relationship between the pins and expansion nodes of the expansion region and the nested activity nodes in the body of the expansion region.

Note, in particular, that an expansion activation group defines output pin activations corresponding to the input pins and expansion nodes of the expansion region. This is to allow these output pin activations to be connected to input pin activations within the expansion activation group. Tokens are placed on the output pin activations for input values to be sent into the group and they then flow to the appropriate input pins within the group via the normal token/offer semantics.

An expansion activation group also defines output pin activations on which the outputs of the group are placed, corresponding to the output expansion nodes of the expansion region. (An expansion region in fUML is not allowed to have output pins – see 7.11 for more information.)

Since an expansion region is syntactically a kind of structured activity node, it includes the option of running its body in isolation (i.e., with mustIsolate = true). However, since ExpansionRegionActivation does not specialize StructuredActivityNodeActivation, it does not automatically inherit the behavior defined in StructuredActivityNodeActivation for isolation (see above). Nevertheless, the class ExpansionRegionActivation uses a similar pattern to StructuredActivityActivation to handle isolation. That is, ExpansionRegionActivation::doAction checks whether mustIsolate = true for the associated expansion region and, if so, it calls ExpansionRegionActivation::doStructuredActivity within a structured activity node with mustIsolate = true. Otherwise it calls doStructuredActivity with no isolation.

Other Actions

Finally, the fUML subset also includes the reduce and raise exception actions.

The reduce action calls a reducer behavior repeatedly in order to reduce a set of input values to a single value. Similarly to a call action, the reduce action activation creates an execution object for the reducer behavior (see Error: Reference source not found). A new execution object is created for each call and is destroyed at the end of the call.

The raise exception action takes a single value as input and raises it as an exception. The raise exception activation does this by simply propagating the exception using the general propagateException operation inherited from ActionActivation.

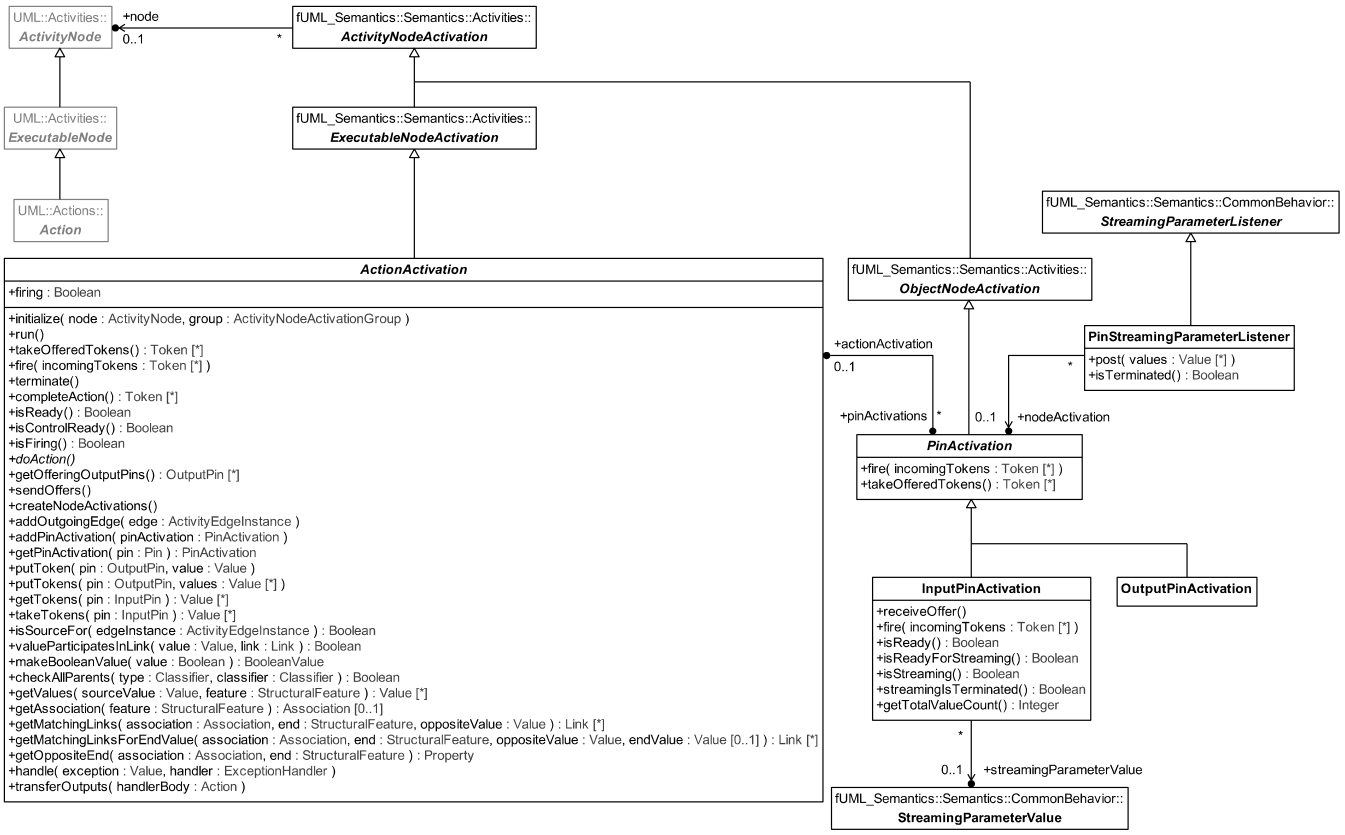

UML::Activities::
ActivityNode
+node
0..1
*
fUML_Semantics::Semantics::Activities::
ActivityNodeActivation
UML::Activities::
ExecutableNode
fUML_Semantics::Semantics::Activities::
ExecutableNodeActivation
UML::Actions::
Action
fUML_Semantics::Semantics::CommonBehavior::
StreamingParameterListener
ActionActivation
+firing : Boolean
+initialize(node : ActivityNode, group : ActivityNodeActivationGroup)
+run()
+takeOfferedTokens() : Token [*]
+fire(incomingTokens : Token [*])
+terminate()
+completeAction() : Token [*]
+isReady() : Boolean
+isControlReady() : Boolean
+isFiring() : Boolean
+doAction()
+getOfferingOutputPins() : OutputPin [*]
+sendOffers()
+createNodeActivations()
+addOutgoingEdge(edge : ActivityEdgeInstance)
+addPinActivation(pinActivation : PinActivation)
+getPinActivation(pin : Pin) : PinActivation
+putToken(pin : OutputPin, value : Value)
+putTokens(pin : OutputPin, values : Value [*])
+getTokens(pin : InputPin) : Value [*]
+takeTokens(pin : InputPin) : Value [*]
+isSourceFor(edgeInstance : ActivityEdgeInstance) : Boolean
+valueParticipatesInLink(value : Value, link : Link) : Boolean
+makeBooleanValue(value : Boolean) : BooleanValue
+checkAllParents(type : Classifier, classifier : Classifier) : Boolean
+getValues(sourceValue : Value, feature : StructuralFeature) : Value [*]
+getAssociation(feature : StructuralFeature) : Association [0..1]
+getMatchingLinks(association : Association, end : StructuralFeature, oppositeValue : Value) : Link [*]
+getMatchingLinksForEndValue(association : Association, end : StructuralFeature, oppositeValue : Value, endValue : Value [0..1]) : Link [*]
+getOppositeEnd(association : Association, end : StructuralFeature) : Property
+handle(exception : Value, handler : ExceptionHandler)
+transferOutputs(handlerBody : Action)
fUML_Semantics::Semantics::Activities::
ObjectNodeActivation
PinStreamingParameterListener
+post(values : Value [*])
+isTerminated() : Boolean
+actionActivation
0..1
*
+pinActivations
*
0..1
+nodeActivation
PinActivation
+fire(incomingTokens : Token [*])
+takeOfferedTokens() : Token [*]
InputPinActivation
+receiveOffer()
+fire(incomingTokens : Token [*])
+isReady() : Boolean
+isReadyForStreaming() : Boolean
+isStreaming() : Boolean
+streamingIsTerminated() : Boolean
+getTotalValueCount() : Integer
OutputPinActivation
*
0..1
+streamingParameterValue
fUML_Semantics::Semantics::CommonBehavior::
StreamingParameterValue

Figure 8.27 - Action Activations

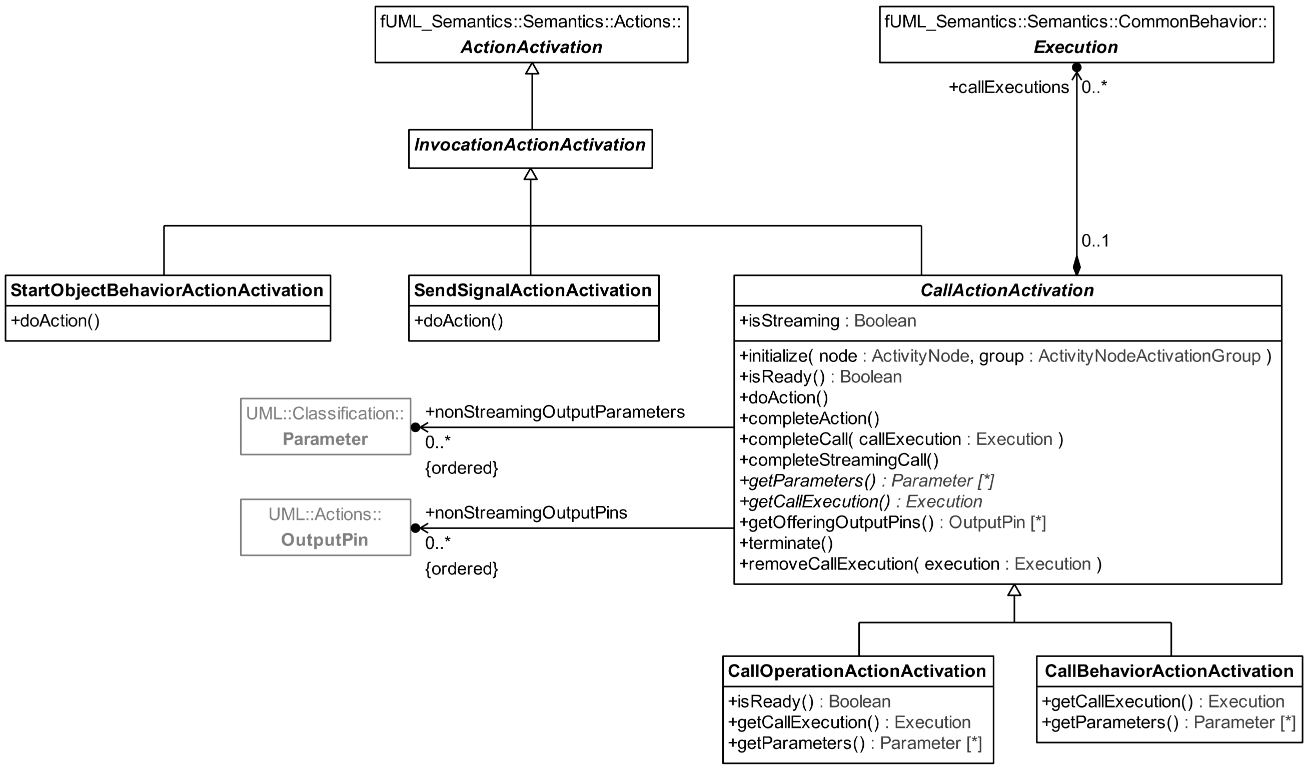
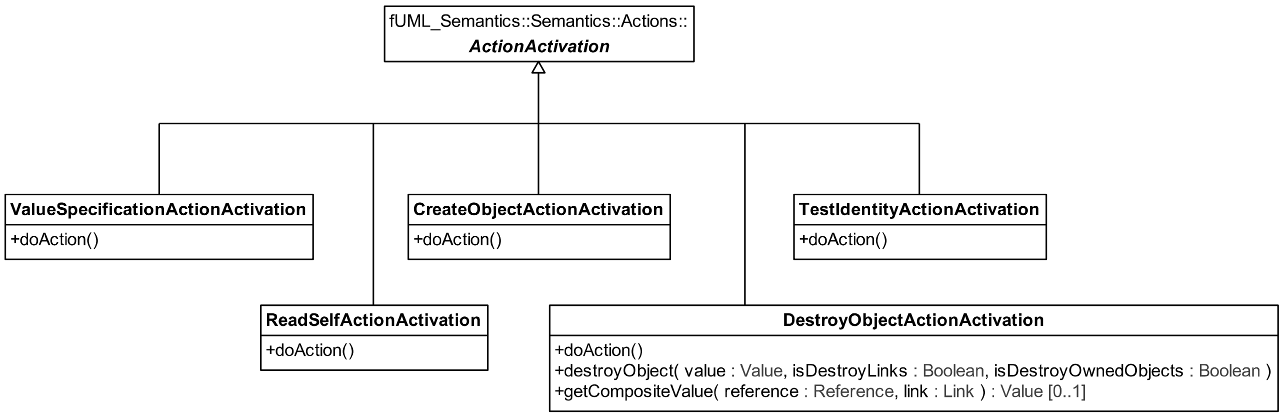

fUML_Semantics::Semantics::Actions::
ActionActivation
fUML_Semantics::Semantics::CommonBehavior::
Execution
+callExecutions
0..*
InvocationActionActivation
0..1
StartObjectBehaviorActionActivation
+doAction()
SendSignalActionActivation
+doAction()
CallActionActivation
+isStreaming : Boolean
+initialize(node : ActivityNode, group : ActivityNodeActivationGroup)
+isReady() : Boolean
+doAction()
+completeAction()
+completeCall(callExecution : Execution)
+completeStreamingCall()
+getParameters() : Parameter [*]
+getCallExecution() : Execution
+getOfferingOutputPins() : OutputPin [*]
+terminate()
+removeCallExecution(execution : Execution)
UML::Classification::
Parameter
+nonStreamingOutputParameters
0..*
{ordered}
UML::Actions::
OutputPin
+nonStreamingOutputPins
0..*
{ordered}
CallOperationActionActivation
+isReady() : Boolean
+getCallExecution() : Execution
+getParameters() : Parameter [*]
CallBehaviorActionActivation
+getCallExecution() : Execution
+getParameters() : Parameter [*]

Figure 8.28 - Invocation Action Activations


fUML_Semantics::Semantics::Actions::
ActionActivation
ValueSpecificationActionActivation
+doAction()
CreateObjectActionActivation
+doAction()
TestIdentityActionActivation
+doAction()
ReadSelfActionActivation
+doAction()
DestroyObjectActionActivation
+doAction()
+destroyObject(value : Value, isDestroyLinks : Boolean, isDestroyOwnedObjects : Boolean)
+getCompositeValue(reference : Reference, link : Link) : Value [0..1]

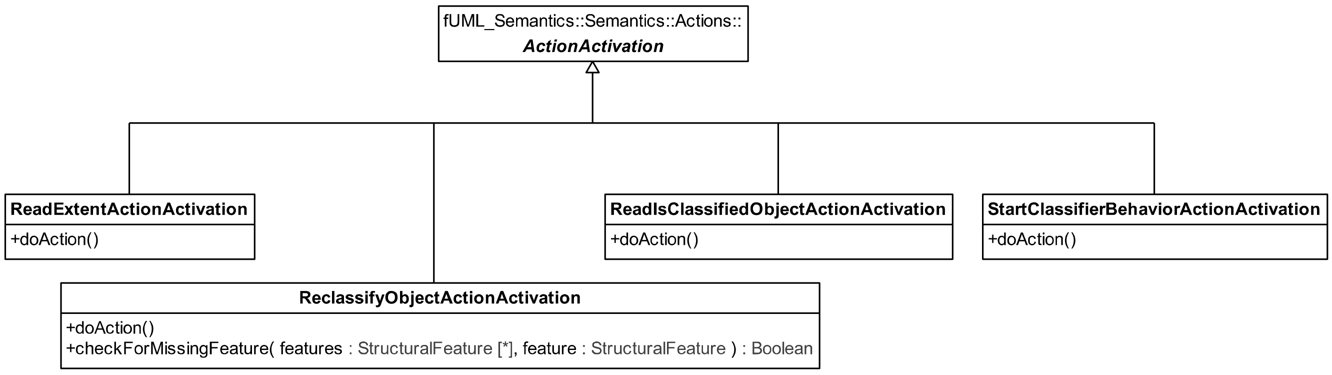

fUML_Semantics::Semantics::Actions::
ActionActivation
ReadExtentActionActivation
+doAction()
ReadIsClassifiedObjectActionActivation
+doAction()
StartClassifierBehaviorActionActivation
+doAction()
ReclassifyObjectActionActivation
+doAction()
+checkForMissingFeature(features : StructuralFeature [*], feature : StructuralFeature) : Boolean

Figure 8.29 - Object Action Activations

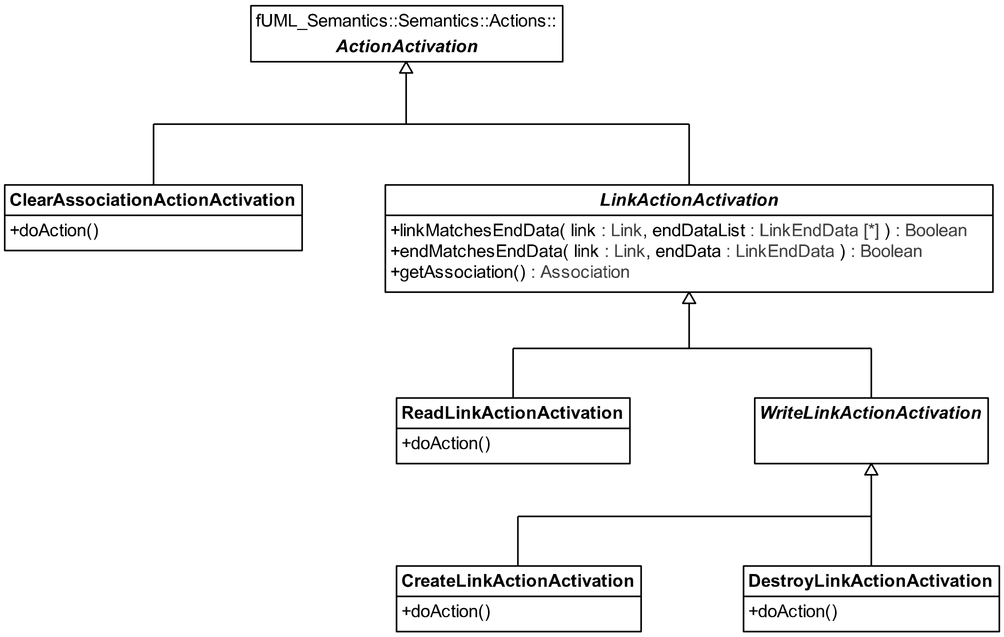

fUML_Semantics::Semantics::Actions::
ActionActivation
ClearAssociationActionActivation
+doAction()
LinkActionActivation
+linkMatchesEndData(link : Link, endDataList : LinkEndData [*]) : Boolean
+endMatchesEndData(link : Link, endData : LinkEndData) : Boolean
+getAssociation() : Association
ReadLinkActionActivation
+doAction()
WriteLinkActionActivation
CreateLinkActionActivation
+doAction()
DestroyLinkActionActivation
+doAction()

Figure 8.30 - Link Action Activations

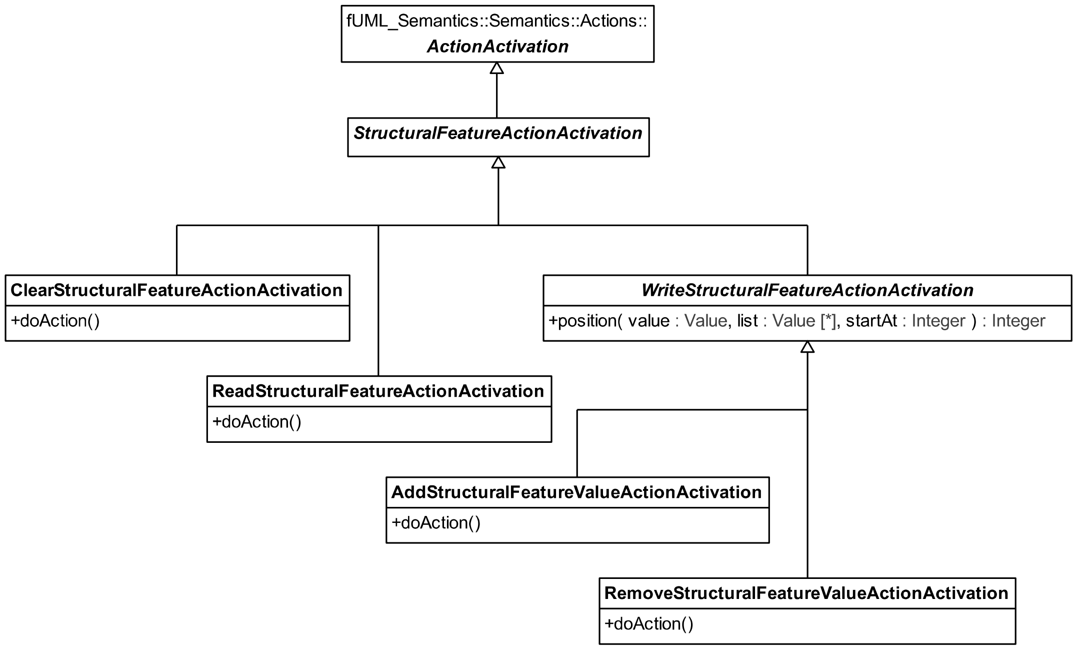

fUML_Semantics::Semantics::Actions::
ActionActivation
StructuralFeatureActionActivation
ClearStructuralFeatureActionActivation
+doAction()
WriteStructuralFeatureActionActivation
+position(value : Value, list : Value [*], startAt : Integer) : Integer
ReadStructuralFeatureActionActivation
+doAction()
AddStructuralFeatureValueActionActivation
+doAction()
RemoveStructuralFeatureValueActionActivation
+doAction()

Figure 8.31 - Structural Feature Action Activations

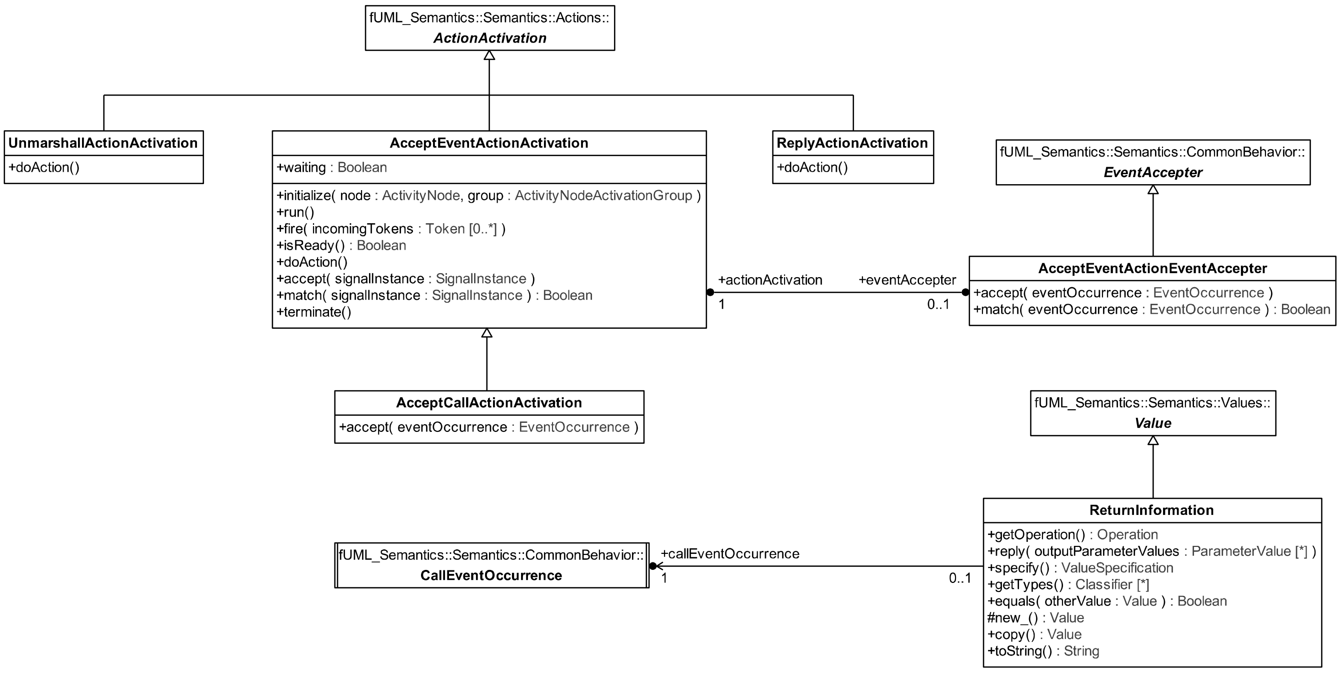
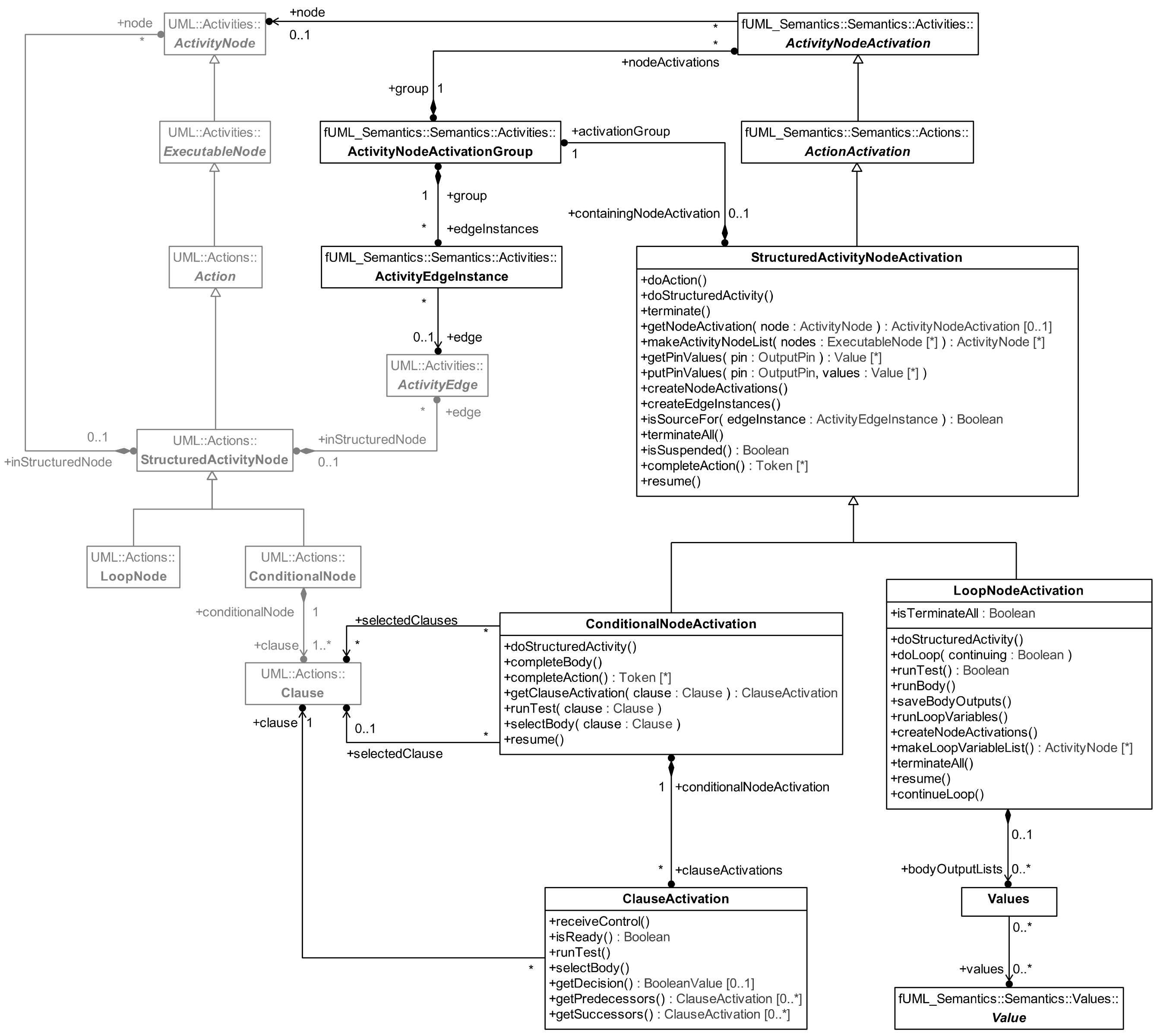

fUML_Semantics::Semantics::Actions::
ActionActivation
UnmarshallActionActivation
+doAction()
AcceptEventActionActivation
+waiting : Boolean
+initialize(node : ActivityNode, group : ActivityNodeActivationGroup)
+run()
+fire(incomingTokens : Token [0..*])
+isReady() : Boolean
+doAction()
+accept(signalInstance : SignalInstance)
+match(signalInstance : SignalInstance) : Boolean
+terminate()
ReplyActionActivation
+doAction()
fUML_Semantics::Semantics::CommonBehavior::
EventAccepter
+actionActivation
1
+eventAccepter
0..1
AcceptEventActionEventAccepter
+accept(eventOccurrence : EventOccurrence)
+match(eventOccurrence : EventOccurrence) : Boolean
AcceptCallActionActivation
+accept(eventOccurrence : EventOccurrence)
fUML_Semantics::Semantics::Values::
Value
ReturnInformation
+getOperation() : Operation
+reply(outputParameterValues : ParameterValue [*])
+specify() : ValueSpecification
+getTypes() : Classifier [*]
+equals(otherValue : Value) : Boolean
#new_() : Value
+copy() : Value
+toString() : String
fUML_Semantics::Semantics::CommonBehavior::
CallEventOccurrence
+callEventOccurrence
1
0..1

Figure 8.32 - Accept Action Activations



UML::Activities:: ActivityNode
+node
*
+node
0..1
fUML_Semantics::Semantics::Activities:: ActivityNodeActivation
*
*
+nodeActivations
+group
1
UML::Activities:: ExecutableNode
fUML_Semantics::Semantics::Activities:: ActivityNodeActivationGroup
+activationGroup
1
fUML_Semantics::Semantics::Actions:: ActionActivation
1
+group
*
+edgeInstances
+containingNodeActivation
0..1
UML::Actions:: Action
fUML_Semantics::Semantics::Activities:: ActivityEdgeInstance
*
0..1
+edge
UML::Activities:: ActivityEdge
*
+edge
StructuredActivityNodeActivation
+doAction()
+doStructuredActivity()
+terminate()
+getNodeActivation(node : ActivityNode) : ActivityNodeActivation [0..1]
+makeActivityNodeList(nodes : ExecutableNode [*]) : ActivityNode [*]
+getPinValues(pin : OutputPin) : Value [*]
+putPinValues(pin : OutputPin, values : Value [*])
+createNodeActivations()
+createEdgeInstances()
+isSourceFor(edgeInstance : ActivityEdgeInstance) : Boolean
+terminateAll()
+isSuspended() : Boolean
+completeAction() : Token [*]
+resume()
0..1
+inStructuredNode
UML::Actions:: StructuredActivityNode
+inStructuredNode
0..1
UML::Actions:: LoopNode
UML::Actions:: ConditionalNode
+conditionalNode
1
+clause
1..*
*
+selectedClauses
*
UML::Actions:: Clause
+clause
1
0..1
*
+selectedClause
ConditionalNodeActivation
+doStructuredActivity()
+completeBody()
+completeAction() : Token [*]
+getClauseActivation(clause : Clause) : ClauseActivation
+runTest(clause : Clause)
+selectBody(clause : Clause)
+resume()
1
+conditionalNodeActivation
*
+clauseActivations
ClauseActivation
+receiveControl()
+isReady() : Boolean
+runTest()
+selectBody()
+getDecision() : BooleanValue [0..1]
+getPredecessors() : ClauseActivation [0..*]
+getSuccessors() : ClauseActivation [0..*]
*
LoopNodeActivation
+isTerminateAll : Boolean
+doStructuredActivity()
+doLoop(continuing : Boolean)
+runTest() : Boolean
+runBody()
+saveBodyOutputs()
+runLoopVariables()
+createNodeActivations()
+makeLoopVariableList() : ActivityNode [*]
+terminateAll()
+resume()
+continueLoop()
0..1
+bodyOutputLists
0..*
Values
0..*
+values
0..*
fUML_Semantics::Semantics::Values:: Value

Figure 8.33 - Structured Action Activations

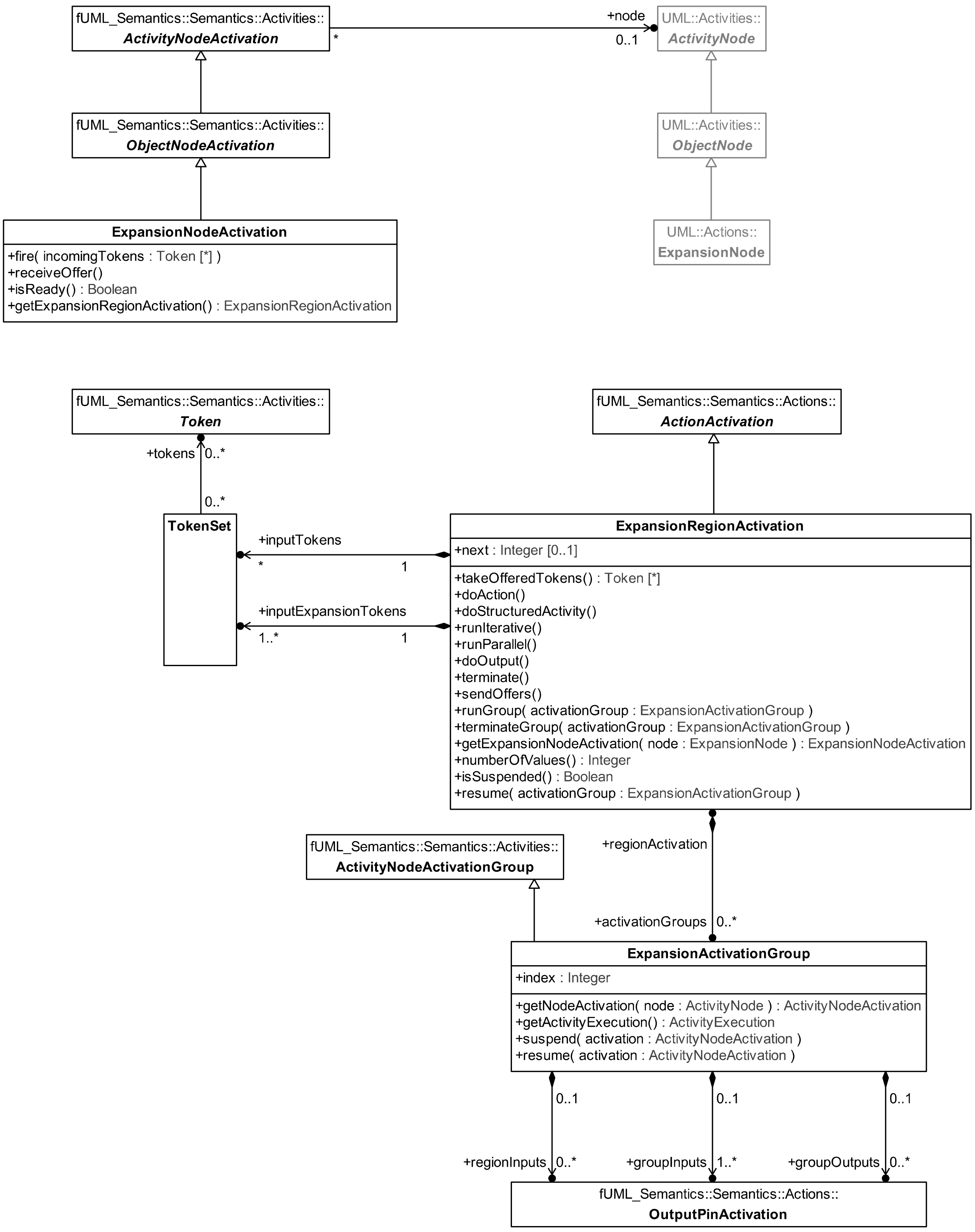

fUML_Semantics::Semantics::Activities::
ActivityNodeActivation
*
+node
0..1
UML::Activities::
ActivityNode
fUML_Semantics::Semantics::Activities::
ObjectNodeActivation
UML::Activities::
ObjectNode
ExpansionNodeActivation
+fire(incomingTokens : Token [*])
+receiveOffer()
+isReady() : Boolean
+getExpansionRegionActivation() : ExpansionRegionActivation
UML::Actions::
ExpansionNode
fUML_Semantics::Semantics::Activities::
Token
+tokens
0..*
0..*
TokenSet
+inputTokens
*
1
+inputExpansionTokens
1..*
1
fUML_Semantics::Semantics::Actions::
ActionActivation
ExpansionRegionActivation
+next : Integer [0..1]
+takeOfferedTokens() : Token [*]
+doAction()
+doStructuredActivity()
+runIterative()
+runParallel()
+doOutput()
+terminate()
+sendOffers()
+runGroup(activationGroup : ExpansionActivationGroup)
+terminateGroup(activationGroup : ExpansionActivationGroup)
+getExpansionNodeActivation(node : ExpansionNode) : ExpansionNodeActivation
+numberOfValues() : Integer
+isSuspended() : Boolean
+resume(activationGroup : ExpansionActivationGroup)
fUML_Semantics::Semantics::Activities::
ActivityNodeActivationGroup
+regionActivation
+activationGroups
0..*
ExpansionActivationGroup
+index : Integer
+getNodeActivation(node : ActivityNode) : ActivityNodeActivation
+getActivityExecution() : ActivityExecution
+suspend(activation : ActivityNodeActivation)
+resume(activation : ActivityNodeActivation)
0..1
0..1
0..1
+regionInputs
0..*
+groupInputs
1..*
+groupOutputs
0..*
fUML_Semantics::Semantics::Actions::
OutputPinActivation

Figure 8.34 - Expansion Region Activations

8.10.2 Class Descriptions

8.10.2.1 AcceptCallActionActivations

An accept call action activation is a specialized accept event action activation for an accept call action.

Generalizations

- AcceptEventActionActivation

Attributes

- None

Associations

- None

Operations

[1] accept (in eventOccurrence : EventOccurrence)

```
// Accept the given event occurrence, which must be a call event occurrence.
// Place return information for the call on the return information
// output pin. Then complete the acceptance of the event occurrence
// as usual.

AcceptCallAction action = (AcceptCallAction) this.node;
OutputPin returnInformationPin = action.returnInformation;

ReturnInformation returnInformation = new ReturnInformation();
returnInformation.callEventOccurrence = (CallEventOccurrence) eventOccurrence;

this.putToken(returnInformationPin, returnInformation);

super.accept(eventOccurrence);
```

8.10.2.2 AcceptEventActionActivation

An accept event action activation is an action activation for an accept event action.

Generalizations

- ActionActivation

Attributes

- waiting : Boolean

Associations

- eventAccepter : AcceptEventActionEventAccepter [0..1]
  If the accept event action activation is waiting for an event, then this is the accepter it has registered for the event.

Operations

[1] accept (in eventOccurrence : EventOccurrence)

```
// Accept the given event occurrence.
// If the action does not unmarshall, then, if the event occurrence is
// a signal event occurrence, place the signal instance of the signal
// event occurrence on the result pin, if any.
// If the action does unmarshall, then get the parameter values of the
// event occurrence, and place the values for each parameter on the
// corresponding output pin.
// Concurrently fire all output pins while offering a single control token.
// If there are no incoming edges, then re-register this accept event action
// execution with the context object.

AcceptEventAction action = (AcceptEventAction)(this.node);
OutputPinList resultPins = action.result;

if (this.running) {
     if (!action.isUnmarshall) {
          if (eventOccurrence instanceof SignalEventOccurrence) {
               SignalInstance signalInstance =
                    ((SignalEventOccurrence)eventOccurrence).signalInstance;
               ValueList result = new ValueList();
               result.addValue(signalInstance);
               if (resultPins.size() > 0) {
                    this.putTokens(resultPins.getValue(0), result);
               }
          }
     } else {
          ParameterValueList parameterValues =
                    eventOccurrence.getParameterValues(action.trigger.get(0).event);
          for (int i = 0; i < parameterValues.size(); i++) {
               ParameterValue parameterValue = parameterValues.getValue(i);
               OutputPin resultPin = resultPins.getValue(i);
               this.putTokens(resultPin, parameterValue.values);
          }
     }

     this.sendOffers();

     this.waiting = false;

     this.receiveOffer();
     this.resume();
}
```

[2] doAction ( )

```
// Do nothing. [This will never be called.]

return;
```

[3] fire (in incomingTokens : Token [0..*])

```
// Register the event accepter for this accept event action activation with the context object of the enclosing activity execution
// and wait for an event to be accepted.

this.getExecutionContext().register(this.eventAccepter);
this.waiting = true;
this.firing = false;

this.suspend();
```

[4] initialize (in node : ActivityNode, in group : ActivityNodeActivationGroup)

```
// Initialize this accept event action activation to be not waiting for an event.

super.initialize(node, group);
this.waiting = false;
```

[5] isReady ( ) : Boolean

```
// An accept event action activation is ready to fire only if it is not already waiting for an event.

boolean ready;
if (this.waiting) {
    ready = false;
} else {
    ready = super.isReady();
}

return ready;
```

[6] match (in eventOccurrence : EventOccurrence) : Boolean

```
// Return true if the given event occurrence matches a trigger of the
// accept event action of this activation.

AcceptEventAction action = (AcceptEventAction) (this.node);
TriggerList triggers = action.trigger;

return  eventOccurrence.matchAny(triggers);
```

[7] run ( )

```
// Create an event accepter and initialize waiting to false.

super.run();

this.eventAccepter = new AcceptEventActionEventAccepter();
this.eventAccepter.actionActivation = this;

this.waiting = false;
```

[8] terminate ( )

```
// Terminate this action and unregister its event accepter.

super.terminate();

if (this.waiting) {
     this.getExecutionContext().unregister(this.eventAccepter);
    this.waiting = false;
}
```

8.10.2.3 AcceptEventActionEventAccepter

An accept event action event accepter handles signal reception events on the behalf of a specific accept event action activation.

Generalizations

- EventAccepter

Attributes

None

Associations

- actionActivation : AcceptEventActionActivation
  The accept event action activation on behalf of which this event accepter is waiting.

Operations

[1] accept (in eventOccurrence : EventOccurrence)

```
// Accept an event occurrence and forward it to the action activation.

this.actionActivation.accept(eventOccurrence);
```

[2] match (in eventOccurrence : EventOccurrence) : Boolean

```
// Return true if the given event occurrence matches a trigger of the accept event
// action of the action activation.

return this.actionActivation.match(eventOccurrence);
```

8.10.2.4 ActionActivation

An action activation is an activity node activation for a node that is an action.

Generalizations

- ActivityNodeActivationExecutableNodeActivation

Attributes

- firing : Boolean
  Whether this action activation is already firing. This attribute is only used if the action for this action activation has isLocallyReentrant = false (the default). If isLocallyReentrant=true, then firing always just remains false.

Associations

- pinActivations : PinActivation [0..*]
  The activations of the pins owned by the action of this action activation.

Operations

[1] addOutgoingEdge (in edge : ActivityEdgeInstance)

```
// If there are no outgoing activity edge instances, create a single activity edge instance
with a fork node execution at the other end.
// Add the give edge to the fork node execution that is the target of the activity edge
instance out of this action execution.
```

```
// [This assumes that all edges directly outgoing from the action are control flows, with an
implicit fork for offers out of the action.]

ActivityNodeActivation  forkNodeActivation;

if (this.outgoingEdges.size() == 0) {
     forkNodeActivation = new ForkNodeActivation();
     forkNodeActivation.running = false;
     ActivityEdgeInstance newEdge = new ActivityEdgeInstance();
     super.addOutgoingEdge(newEdge);
     forkNodeActivation.addIncomingEdge(newEdge);
}
else {
     forkNodeActivation = this.outgoingEdges.getValue(0).target;
}

forkNodeActivation.addOutgoingEdge(edge);
```

[2] addPinActivation (in pinActivation : PinActivation)

```
// Add a pin activation to this action activation.

this.pinActivations.addValue(pinActivation);
pinActivation.actionActivation = this;
```

[3] completeAction ( ) : Token [0..*]

```
// Concurrently fire all output pin activations and offer a single
// control token. Then check if the action should fire again
// and, if so, return additional incoming tokens for this.

this.sendOffers();

_beginIsolation();
TokenList incomingTokens = new TokenList();
this.firing = false;
if (this.isReady()) {
     incomingTokens = this.takeOfferedTokens();
     this.firing = this.isFiring() & incomingTokens.size() > 0;
}
_endIsolation();
```

```
return incomingTokens;
```

[4] createNodeActivations ( )

```
// Create node activations for the input and output pins of the action for this activation.
// [Note: Pins are owned by their actions, not by the enclosing activity (or group), so they
must be activated through the action activation.]

Action action = (Action)(this.node);

ActivityNodeList inputPinNodes = new ActivityNodeList();
InputPinList inputPins = action.input;
for (int i = 0; i < inputPins.size(); i++) {
     InputPin inputPin = inputPins.getValue(i);
     inputPinNodes.addValue(inputPin);
}

this.group.createNodeActivations(inputPinNodes);

for (int i = 0; i < inputPinNodes.size(); i++) {
     ActivityNode node = inputPinNodes.getValue(i);
      this.addPinActivation((PinActivation)(this.group.getNodeActivation(node)));
}

ActivityNodeList outputPinNodes = new ActivityNodeList();
OutputPinList outputPins = action.output;
for (int i = 0; i < outputPins.size(); i++) {
     OutputPin outputPin = outputPins.getValue(i);
     outputPinNodes.addValue(outputPin);
}

this.group.createNodeActivations(outputPinNodes);

for (int i = 0; i < outputPinNodes.size(); i++) {
     ActivityNode node = outputPinNodes.getValue(i);
      this.addPinActivation((PinActivation)(this.group.getNodeActivation(node)));
}
```

[5] doAction ( )

```
Do the required action behavior.
```

[6] fire (in incomingTokens : Token [0..*])

```
// Do the main action behavior then concurrently fire all output pin activations
// and offer a single control token. Then activate the action again,
// if it is still ready to fire and has at least one token actually being
// offered to it.

do {

     this.doAction();
     incomingTokens = this.completeAction();

} while (incomingTokens.size() > 0);
```

[7] getAssociation (in feature : StructuralFeature) : Association [0..1]

```
// If the given structural feature is an association end, then get
// the associated association.

Association association = null;
if (feature instanceof Property) {
    association = ((Property)feature).association;
}

return association;
```

[8] getMatchingLinks (in association : Association, in end : StructuralFeature, in oppositeValue : Value) : Link [0..*]

```
// Get the links of the given binary association whose end opposite
// to the given end has the given value

return this.getMatchingLinksForEndValue(association, end, oppositeValue, null);
```

[9] getMatchingLinksForEndValue (in association : Association, in end : StructuralFeature, in oppositeValue : Value, in endValue : Value [0..1]) : Link [0..*]

```
// Get the links of the given binary association whose end opposite
// to the given end has the given opposite value and, optionally, that
// has a given end value for the given end.

Property oppositeEnd = this.getOppositeEnd(association, end);

ExtensionalValueList extent = this.getExecutionLocus().getExtent(association);
```

```
LinkList links = new LinkList();
for (int i = 0; i<extent.size(); i++) {
    ExtensionalValue link = extent.getValue(i);
    if (link.getFeatureValue(oppositeEnd).values.getValue(0).equals(oppositeValue)) {
        boolean matches = true;
        if (endValue != null) {
          matches = link.getFeatureValue(end).values.getValue(0).equals(endValue);
        }
        if (matches) {
            if (!end.multiplicityElement.isOrdered | links.size() == 0) {
                links.addValue((Link)link);
            } else {
                int n = link.getFeatureValue(end).position;
                boolean continueSearching = true;
                int j = 0;
                while (continueSearching & j < links.size()) {
                    j = j + 1;
                    continueSearching =
                             links.getValue(j-1).getFeatureValue(end).position < n;
                }
                if (continueSearching) {
                    links.addValue((Link)link);
                } else {
                    links.addValue(j-1, (Link)link);
                }
            }
        }
    }
}

return links;
```

[10] getOppositeEnd (in association : Association, in end : StructuralFeature) : Property

```
// Get the end of a binary association opposite to the given end.

Property oppositeEnd = association.memberEnd.getValue(0);
if (oppositeEnd == end) {
   oppositeEnd = association.memberEnd.getValue(1);
}
```

```
return oppositeEnd;
```

[11] getPinActivation (in pin : Pin) : PinActivation

```
// Precondition: The given pin is owned by the action of the action activation.
// Return the pin activation corresponding to the given pin.

PinActivation pinActivation = null;
int i = 1;
while (pinActivation == null & i <= this.pinActivations.size()) {
     PinActivation thisPinActivation = this.pinActivations.getValue(i-1);
     if (thisPinActivation.node == pin) {
          pinActivation = thisPinActivation;
     }
     i = i + 1;
}

return pinActivation;
```

[12] getTokens (in pin : InputPin) : Value [0..*]

```
// Precondition: The action execution has fired and the given pin is owned by the action of
the action execution.
// Get any tokens held by the pin activation corresponding to the given input pin and return
them
// (but leave the tokens on the pin).

PinActivation pinActivation = this.getPinActivation(pin);
TokenList tokens = pinActivation.getUnofferedTokens();

ValueList values = new ValueList();
for (int i = 0; i < tokens.size(); i++) {
     Token token = tokens.getValue(i);
     Value value = ((ObjectToken)token).value;
     if (value != null) {
          values.addValue(value);
     }
}

return values;
```

[13] getOfferingOutputPins ( ) : OutputPin [0..*]

```
// Return the output pins of the action of this action activation from
// which offers are to be sent when the action activation finishes firing.
// (This is normally all the output pins of the action, but it can be
// overridden in subclasses to only return a subset of the output pins.)

return ((Action)this.node).output;
```

[14] getValues (in sourceValue : Value, in feature : StructuralFeature) : Value [0..*]

```
// Get the values of the feature for the given source value.
// If the feature is an association end, then get the values of
// the feature end of the links with the source value as the
// opposite end.
// Otherwise, if the source value is a structured value, get
// the values of the feature value for feature in the structured value.

ValueList values = new ValueList();

Association association = this.getAssociation(feature);
if (association != null) {
    LinkList links = this.getMatchingLinks(association, feature, sourceValue);
    for (int j = 0; j < links.size(); j++) {
        Link link = links.getValue(j);
        values.addValue(link.getFeatureValue(feature).values.getValue(0));
    }
} else {
    values = ((StructuredValue)sourceValue).getFeatureValue(feature).values;
}

return values;
```

[15] handle (in exception : Value, in handler : ExceptionHandler)

```
// Handle the given exception by firing the body of the given
// exception handler. After the body fires, transfer its outputs
// to the output pins of this action activation.

super.handle(exception, handler);
this.transferOutputs((Action)handler.handlerBody);
```

[16] initialize (in node : ActivityNode, in group : ActivityNodeActivationGroup)

```
// Initialize this action activation to be not firing.
```

```
super.initialize(node, group);
this.firing = false;
```

[17] isControlReady ( ) : Boolean

```
// In addition to the default condition for being ready, check that,
// if the action has isLocallyReentrant=false, then the activation is
// not currently firing, and that the sources of all incoming edges
// have offers. (This assumes that all edges directly incoming to the
// action are control flows.)

boolean ready = super.isReady()
        & (((Action) this.node).isLocallyReentrant | !this.isFiring());

int i = 1;
while (ready & i <= this.incomingEdges.size()) {
        ready = this.incomingEdges.getValue(i - 1).hasOffer();
        i = i + 1;
}

return ready;
```

[18] isFiring ( ) : Boolean

```
// Indicate whether this action activation is currently firing or not.

return  firing;
```

[19] isReady ( ) : Boolean

// Check that the action is ready to fire, including

```
// that all input pin activations are ready.

boolean ready = isControlReady();

InputPinList inputPins = ((Action) (this.node)).input;
int j = 1;
while (ready & j <= inputPins.size()) {
    ready = this.getPinActivation(inputPins.getValue(j - 1)).isReady();
    j = j + 1;
}
```

```
return ready;
```

[20] isSourceFor (in edgeInstance : ActivityEdgeInstance) : Boolean

```
// If this action has an outgoing fork node, check that the fork node is the source of the
given edge instance.

boolean isSource = false;
if (this.outgoingEdges.size() > 0) {
     isSource = this.outgoingEdges.getValue(0).target.isSourceFor(edgeInstance);
}

return isSource;
```

[21] makeBooleanValue (in value : Boolean) : BooleanValue

```
// Make a Boolean value using the built-in Boolean primitive type.
// [This ensures that Boolean values created internally are the same as the default used for
evaluating Boolean literals.]

LiteralBoolean booleanLiteral = new LiteralBoolean();
booleanLiteral.value = value;
return  (BooleanValue)(this.getExecutionLocus().executor.evaluate(booleanLiteral));
```

[22] putToken (in pin : OutputPin, in value : Value)

```
// Precondition: The action execution has fired and the given pin is owned by the action of
the action execution.
// Place a token for the given value on the pin activation corresponding to the given output
pin.

ObjectToken token = new ObjectToken();
token.value = value;

PinActivation pinActivation = this.getPinActivation(pin);
pinActivation.addToken(token);
```

[23] putTokens (in pin : OutputPin, in values : Value [0..*])

```
// Precondition: The action execution has fired and the given pin is owned by the action of
the action execution.
// Place tokens for the given values on the pin activation corresponding to the given output
pin.

for (int i = 0; i < values.size(); i++) {
    Value value = values.getValue(i);
    this.putToken(pin, value);
}
```

[23] run ( )

```
// Run this action activation and any outgoing fork node attached to it.

super.run();

if (this.outgoingEdges.size() > 0) {
    this.outgoingEdges.getValue(0).target.run();
}

this.firing = false;
```

[25] sendOffers ( )

```
// Fire all output pins and send offers on all outgoing control flows.

Action action = (Action)(this.node);

// *** Send offers from all output pins concurrently. ***

OutputPinList outputPins = this.getOfferingOutputPins();
for (Iterator i = outputPins.iterator(); i.hasNext();) {
    OutputPin outputPin = (OutputPin)i.next();
    PinActivation pinActivation = this.getPinActivation(outputPin);
    pinActivation.sendUnofferedTokens();
}

// Send offers on all outgoing control flows.
if (this.outgoingEdges.size() > 0) {
    TokenList tokens = new TokenList();
    tokens.addValue(new ControlToken());
    this.addTokens(tokens);
    this.outgoingEdges.getValue(0).sendOffer(tokens);
}
[24] takeOfferedTokens ( ) : Token [0..*]
// If the action is not locally reentrant, then mark this activation as firing.
// Take any incoming offers of control tokens, then concurrently fire all input pin
activations.
```

```
// Note: This is included here to happen in the same isolation scope as the isReady test.
this.firing = !((Action)this.node).isLocallyReentrant;

TokenList offeredTokens = new TokenList();

ActivityEdgeInstanceList incomingEdges = this.incomingEdges;
for (int i = 0; i < incomingEdges.size(); i++) {
    ActivityEdgeInstance incomingEdge = incomingEdges.getValue(i);
    TokenList tokens = incomingEdge.takeOfferedTokens();
    for (int j = 0; j < tokens.size(); j++) {
        Token token = tokens.getValue(j);
        token.withdraw();
        offeredTokens.addValue(token);
    }
}

Action action = (Action)(this.node);

// *** Fire all input pins concurrently. ***
InputPinList inputPins = action.input;
for (Iterator i = inputPins.iterator(); i.hasNext();) {
    InputPin pin = (InputPin)(i.next());
    PinActivation pinActivation = this.getPinActivation(pin);
    pinActivation.fire(pinActivation.takeOfferedTokens());
    for (int j = 0; j < tokens.size(); j++) {
        Token token = tokens.getValue(j);
        offeredTokens.addValue(token);
    }
}

return offeredTokens;
```

[25] takeTokens (in pin : InputPin) : Value [0..*]

```
// Precondition: The action execution has fired and the given pin is owned by the action of
the action execution.
// Take any tokens held by the pin activation corresponding to the given input pin and return
them.

PinActivation pinActivation = this.getPinActivation(pin);
TokenList tokens = pinActivation.takeUnofferedTokens();
```

```
ValueList values = new ValueList();
for (int i = 0; i < tokens.size(); i++) {
     Token token = tokens.getValue(i);
     Value value = ((ObjectToken)token).value;
     if (value != null) {
          values.addValue(value);
     }
}

return values;
```

[26] terminate ( )

```
// Terminate this action activation and any outgoing fork node attached to it.

super.terminate();

if (this.outgoingEdges.size() > 0) {
     this.outgoingEdges.getValue(0).target.terminate();
}
```

[27] transferOutputs (in handlerBody : Action)

```
// Transfer the output values from activation of the given exception
// handler body to the output pins of this action activation.

ActionActivation handlerBodyActivation =
                (ActionActivation)this.group.getNodeActivation(handlerBody);
OutputPinList sourceOutputs = handlerBody.output;
OutputPinList targetOutputs = ((Action) this.node).output;

for (int i = 0; i < sourceOutputs.size(); i++) {
    OutputPin sourcePin = sourceOutputs.getValue(i);
    OutputPin targetPin = targetOutputs.getValue(i);

    PinActivation sourcePinActivation = handlerBodyActivation.getPinActivation(sourcePin);
    TokenList tokens = sourcePinActivation.takeTokens();
    ValueList values = new ValueList();
    for (int j = 0; j < tokens.size(); j++) {
        Token token = tokens.getValue(j);
        values.addValue(token.getValue());
    }

    this.putTokens(targetPin, values);
}
```

[28] valueParticipatesInLink (in value : Value, in link : Link) : Boolean

```
// Test if the given value participates in the given link.

FeatureValueList linkFeatureValues = link.getFeatureValues();

boolean participates = false;
int i = 1;
while (!participates & i <= linkFeatureValues.size()) {
     participates = linkFeatureValues.getValue(i-1).values.getValue(0).equals(value);
    i = i + 1;
}

return participates;
```

8.10.2.5 AddStructuralFeatureValueActionActivation

An add structural feature action value activation is a write structural feature action activation for an add structural feature value action.

Generalizations

- WriteStructuralFeatureActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the values of the object and value input pins.
// If the given feature is an association end, then create a link between the object and
value inputs.
// Otherwise, if the object input is a structural value, then add a value to the values for
the feature.
// If isReplaceAll is true, first remove all current matching links or feature values.
// If isReplaceAll is false and there is an insertAt pin, insert the value at the appropriate
position.
```

```
AddStructuralFeatureValueAction action = (AddStructuralFeatureValueAction)(this.node);
StructuralFeature feature = action.structuralFeature;
Association association = this.getAssociation(feature);

Value value = this.takeTokens(action.object).getValue(0);
ValueList inputValues = this.takeTokens(action.value);

// NOTE: Multiplicity of the value input pin is required to be 1..1.
Value inputValue = inputValues.getValue(0);

int insertAt = 0;
if (action.insertAt != null) {
    insertAt =
((UnlimitedNaturalValue)this.takeTokens(action.insertAt).getValue(0)).value.naturalValue;
}

if (association != null) {
     LinkList links = this.getMatchingLinks(association, feature, value);

     Property oppositeEnd = this.getOppositeEnd(association, feature);
     int position = 0;
     if (oppositeEnd.multiplicityElement.isOrdered) {
          position = this.getMatchingLinks(association, oppositeEnd, inputValue).size() + 1;
     }

     if (action.isReplaceAll) {
         for (int i = 0; i < links.size(); i++) {
             Link link = links.getValue(i);
             link.destroy();
         }
     } else if (feature.multiplicityElement.isUnique) {

         int i = 1;
         boolean destroyed = false;
         while (!destroyed & i <= links.size()) {
             Link link = links.getValue(i - 1);
             FeatureValue featureValue = link.getFeatureValue(feature);
             if (featureValue.values.getValue(0).equals(inputValue)) {
                 position = link.getFeatureValue(oppositeEnd).position;
                 link.destroy();
```

```
                destroyed = true;
            }
            i = i + 1;
        }

     Link newLink = new Link();
     newLink.type = association;

     newLink.setFeatureValue(feature, inputValues, insertAt);

     ValueList oppositeValues = new ValueList();
     oppositeValues.addValue(value);
     newLink.setFeatureValue(oppositeEnd, oppositeValues, position);

     newLink.addTo(this.getExecutionLocus());

} else if (value instanceof StructuredValue) {

     // If the value is a data value, then it must be copied before
     // any change is made.
     if (!(value instanceof Reference)) {
          value = value.copy();
     }

     StructuredValue structuredValue = (StructuredValue)value;

     if (action.isReplaceAll) {
          structuredValue.setFeatureValue(feature, inputValues, 0);
     } else {
          FeatureValue featureValue = structuredValue.getFeatureValue(feature);

          if (featureValue.values.size() > 0 & insertAt == 0 ) {
              // *** If there is no insertAt pin, then the structural feature must be
unordered, and the insertion position is immaterial. ***
             insertAt =
((ChoiceStrategy)this.getExecutionLocus().factory.getStrategy("choice")).choose(featureValue.
values.size());
         }

          if (feature.multiplicityElement.isUnique) {
              // Remove any existing value that duplicates the input value
```

```
                int j = position(inputValue, featureValue.values, 1);
                if (j > 0) {
                    featureValue.values.remove(j-1);
                }
            }

            if (insertAt <= 0) {  // Note: insertAt = -1 indicates an unlimited value of "*"
                featureValue.values.addValue(inputValue);
            } else {
                featureValue.values.addValue(insertAt - 1, inputValue);
            }
        }
    }

if (action.result != null) {
    this.putToken(action.result, value);
}
```

8.10.2.6 CallActionActivation

A call action activation is an invocation action activation for a call action.

Generalizations

- InvocationActionActivation

Attributes

- isStreaming : Boolean
  Whether the behavior being invoked has streaming input parameters.
- nonStreamingOutputParameters : Parameter [0..*]
  The set of output parameters of the called behavior that are not streaming parameters.
- nonStreamingOutputPins : OutputPin [0..*]
  The set of output pins corresponding to the non-streaming output parameters of the called behavior.

Associations

- callExecutions : Execution [0..*]
  The set of execution object for currently ongoing calls made through this call action activation.

Operations

[1] completeAction ( )

```
// If this call action activation is not streaming, then complete the action
// normally. Otherwise, complete the action without checking for firing again
// (but keep the call execution running).
```

```
TokenList incomingTokens;
if (this.isStreaming) {
    incomingTokens = new TokenList();
} else {
    incomingTokens = super.completeAction();
}
return incomingTokens;
```

[2] completeCall (in callExecution : Execution)

```
// If the call execution raised an exception, then propagate it. Otherwise,
// copy the values of the non-streaming output parameters of the call execution
// to the corresponding result pin activations of the call action activation and
// destroy the execution.

if(callExecution.exception != null) {
     this.propagateException(callExecution.exception);
} else {
     OutputPinList resultPins = this.nonStreamingOutputPins;
     ParameterList parameters = this.nonStreamingOutputParameters;

     ParameterValueList outputParameterValues = callExecution.
          getOutputParameterValues();

     for (int i = 0; i < resultPins.size(); i++) {
         OutputPin resultPin = resultPins.getValue(i);
         Parameter parameter = parameters.getValue(i);
         for (int j = 0; j < outputParameterValues.size(); j++) {
             ParameterValue outputParameterValue = outputParameterValues.
                     getValue(j);
             if (outputParameterValue.parameter == parameter) {
                 this.putTokens(resultPin, outputParameterValue.values);
             }
         }
     }
}

callExecution.destroy();
this.removeCallExecution(callExecution);
```

[3] completeStreamingCall ( )

```
// Complete a streaming call execution and then complete this call action activation.

if (this.callExecutions.size() > 0) {
    // Note: If the call is streaming, then isLocallyReentrant = false and
    // there should be at most one call execution.
    this.completeCall(this.callExecutions.getValue(0));
    super.completeAction();
}
```

[4] doAction ( )

```
// Get the call execution object, set its input parameters from the
// argument pins and execute it.
// If there are no streaming input parameters, then, once execution completes,
// copy the values of the output parameters of the call execution to the result
// pins of the call action execution and destroy the execution.
// If there are streaming input parameters, then leave the call execution object
// in place to process any additional inputs that may be posted to the streaming
// input parameters.

Execution callExecution = this.getCallExecution();

if (callExecution != null) {
    this.callExecutions.addValue(callExecution);

    CallAction callAction = (CallAction) (this.node);
    InputPinList argumentPins = callAction.argument;
    OutputPinList resultPins = callAction.result;

    // Must get parameters from call execution behavior, to ensure the correct
    // parameters are used for an operation method.
    ParameterList parameters = callExecution.getBehavior().ownedParameter;

    int pinNumber = 1;
    int outputPinNumber = 1;
    int i = 1;
    InputPinActivation streamingPinActivation = null;
    this.nonStreamingOutputPins.clear();
    this.nonStreamingParameters.clear();
```

```
while (i <= parameters.size()) {
    Parameter parameter = parameters.getValue(i - 1);
    if (parameter.direction == ParameterDirectionKind.in
            | parameter.direction == ParameterDirectionKind.inout) {
        InputPin argumentPin = argumentPins.getValue(pinNumber - 1);
        ParameterValue parameterValue;
        if (parameter.isStream) {
            this.isStreaming = true;
            parameterValue = new StreamingParameterValue();
            parameterValue.values = this.getTokens(argumentPin);
            streamingPinActivation =
                (InputPinActivation) this.getPinActivation(argumentPin);
            streamingPinActivation.streamingParameterValue =
                (StreamingParameterValue)parameterValue;
        } else {
            parameterValue = new ParameterValue();
            parameterValue.values = this.takeTokens(argumentPin);
        }
        parameterValue.parameter = parameter;
        callExecution.setParameterValue(parameterValue);
        pinNumber = pinNumber + 1;
    }
    if (parameter.direction == ParameterDirectionKind.out
            | parameter.direction == ParameterDirectionKind.inout
            | parameter.direction == ParameterDirectionKind.return_) {
        OutputPin resultPin = resultPins.getValue(outputPinNumber - 1);
        if (!parameter.isStream) {
            this.nonStreamingOutputPins.addValue(resultPin);
            this.nonStreamingOutputParameters.addValue(parameter);
        } else {
            ParameterValue parameterValue = new StreamingParameterValue();
            parameterValue.parameter = parameter;
            PinStreamingParameterListener listener =
                new PinStreamingParameterListener();
            listener.nodeActivation = this.getPinActivation(resultPin);
            ((StreamingParameterValue)parameterValue).register(listener);

            // Note: Add a new parameter value, so that there will
            // be two separate input and output parameter values for a
            // streaming inout parameter.
            callExecution.parameterValues.addValue(parameterValue);
            }
            outputPinNumber = outputPinNumber + 1;
        }
        i = i + 1;
    }

    callExecution.execute();

    if (streamingPinActivation == null) {
        this.isStreaming = false;
    } else {
        this.isStreaming = !streamingPinActivation.streamingIsTerminated();
    }

    if (!this.isStreaming) {
        this.completeCall(callExecution);
    }
}
```

[5] getCallExecution ( ) : Execution

```
Get the execution object for the called behavior.
```

[6] getOfferingOutputPins ( ) : OutputPin [0..*]

```
// Only send offers from output pins that correspond to non-streaming parameters.

return this.nonStreamingOutputPins;
```

[7] getParameters ( ) : Parameter [0..*]

```
Get the parameters associated with the pins of the call action for this call action activation.
```

[8] initialize (in node : ActivityNode, in group : ActivityNodeActivationGroup)

```
// Initialize this call action activation to be not streaming.

super.initialize(node, group);
this.isStreaming = false;
```

[9] isReady ( ) : Boolean

```
// Check that this call action activation is ready to fire, accounting for
// the possibility of pins corresponding to streaming parameters. In order
// to be ready, only argument pin activations for non-streaming parameters must
// be ready, except if all the argument pin activations are for streaming
// parameters with multiplicity lower bound greater than 0, in which case
// at least one of those pins must have an offered value.

boolean ready = this.isControlReady();

CallAction callAction = (CallAction) (this.node);
InputPinList argumentPins = callAction.argument;

if (ready & argumentPins.size() > 0) {
    ParameterList parameters = this.getParameters();
    ParameterList inputParameters = new ParameterList();
    for (int i = 0; i < parameters.size(); i++) {
        Parameter parameter = parameters.getValue(i);
        if (parameter.direction == ParameterDirectionKind.in
                | parameter.direction == ParameterDirectionKind.inout) {
            inputParameters.addValue(parameter);
        }
    }

    boolean streamingReady = false;
    int j = 1;
    while (ready & j <= argumentPins.size()) {
        InputPin argumentPin = argumentPins.getValue(j - 1);
        InputPinActivation pinActivation =
                (InputPinActivation)this.getPinActivation(argumentPin);
        if (j > inputParameters.size()) {
            ready = pinActivation.isReady();
        }
        boolean isStream = false;
        if (j <= inputParameters.size()) {
            isStream = inputParameters.getValue(j - 1).isStream;
        }
        if (!isStream) {
            // If there are any non-streaming argument pins, then streaming
            // is considered to be ready.
            streamingReady = true;

            // All non-streaming argument pins must be ready.
            ready = pinActivation.isReady();
        } else if (pinActivation.isReadyForStreaming()) {
            // If there are only streaming argument pins, then streaming
            // is ready if any of them are ready for streaming.
            streamingReady = true;
        }
        j = j + 1;
    }

    ready = ready & streamingReady;
}

return ready;
```

[10] removeCallExecution (in execution : Execution)

```
// Remove the given execution from the current list of call executions.

boolean notFound = true;
int i = 1;
while (notFound & i <= this.callExecutions.size()) {
     if (this.callExecutions.getValue(i-1) == execution) {
         this.callExecutions.removeValue(i-1);
         notFound = false;
    }
    i = i + 1;
}
```

[11] terminate ( )

// Terminate all call executions (if any). If this call action

```
// activation is streaming, complete the call before terminating the call
// execution. Finally, terminate the call action activation itself.

if (this.isStreaming) {
    this.completeStreamingCall();
} else {
    for (int i = 0; i < this.callExecutions.size(); i++) {
        Execution execution = this.callExecutions.getValue(i);
        execution.terminate();
    }
}
```

```
super.terminate();
```

8.10.2.7 CallBehaviorActionActivation

A call behavior action activation is a call action activation for a call behavior action.

Generalizations

- CallActionActivation

Attributes

None

Associations

None

Operations

[1] getCallExecution ( ) : Execution

```
// Create and execution for the given behavior at the current locus and return the resulting
execution object.
// If the given behavior is in the context of a classifier, then pass the current context
object as the context for the call.
// Otherwise, use a null context.
// [Note that this requires the behavior context to be compatible with the type of the
current context object.]

Behavior behavior = ((CallBehaviorAction)(this.node)).behavior;

Object_  context;
if (behavior.context == null) {
    context = null;
} else {
     context = this.getExecutionContext();
}

return  this.getExecutionLocus().factory.createExecution(behavior,  context);
```

[2] getParameters ( ) : Parameter [0..*]

```
// Get the owned parameters of the behavior of the call behavior
// action for this call behavior action activation.

return ((CallBehaviorAction) (this.node)).behavior.ownedParameter;
```

8.10.2.8 CallOperationActionActivation

A call operation action activation is a call action activation for a call operation action.

Generalizations

- CallActionActivation

Attributes

None

Associations

None

Operations

[1] getCallExecution ( ) : Execution

```
// If the value on the target input pin is a reference, dispatch the operation to it and
return the resulting execution object.

CallOperationAction action = (CallOperationAction)(this.node);
Value target = this.takeTokens(action.target).getValue(0);

Execution execution;
if (target instanceof Reference) {
     execution = ((Reference)target).dispatch(action.operation);
}
else {
     execution = null;
}

return execution;
```

[2] getParameters ( ) : Parameter [0..*]

```
// Get the owned parameters of the operation of the call operation
// action for this call operation action activation.

return ((CallOperationAction) (this.node)).operation.ownedParameter;
```

[3] isReady ( ) : Boolean

```
// Check that this call operation action activation is ready to fire as a
// call action activation and, in addition, that the input pin activation
// for its target pin is ready to fire.

boolean ready = super.isReady();
if (ready) {
        CallOperationAction action = (CallOperationAction) (this.node);
        ready = this.getPinActivation(action.target).isReady();
}
return ready;
```

8.10.2.9 ClauseActivation

A clause activation defines the behavior of a clause within the context of a specific activation of the conditional node containing the clause.

Generalizations

None

Attributes

None

Associations

- clause : Clause
- conditionalNodeActivation : ConditionalNodeActivation

  The activation of the conditional node that contains the clause for this clause activation.

Operations

[1] getDecision ( ) : BooleanValue [0..1]

```
// Get the value (if any) on the decider pin of the clause for this clause activation.

ValueList deciderValues = this.conditionalNodeActivation.getPinValues(this.clause.decider);

BooleanValue deciderValue = null;
if (deciderValues.size()  > 0) {
     deciderValue = (BooleanValue)(deciderValues.getValue(0));
}

return deciderValue;
```

[2] getPredecessors ( ) : ClauseActivation [0..*]

```
// Return the clause activations for the predecessors of the clause for this clause
activation.

ClauseActivationList predecessors = new ClauseActivationList();
```

```
ClauseList predecessorClauses = this.clause.predecessorClause;
for (int i = 0; i < predecessorClauses.size(); i++) {
     Clause predecessorClause = predecessorClauses.getValue(i);

predecessors.addValue(this.conditionalNodeActivation.getClauseActivation(predecessorClause));
}

return predecessors;
```

[3] getSuccessors ( ) : ClauseActivation [0..*]

```
// Return the clause activations for the successors of the clause for this clause activation.

ClauseActivationList successors = new ClauseActivationList();

ClauseList successorClauses = this.clause.successorClause;
for (int i = 0; i < successorClauses.size(); i++) {
     Clause successorClause = successorClauses.getValue(i);
      successors.addValue(this.conditionalNodeActivation.getClauseActivation(successorClause));
}

return successors;
```

[4] isReady ( ) : Boolean

```
// Test if all predecessors to this clause activation have failed.

ClauseActivationList predecessors = this.getPredecessors();

boolean ready = true;
int i = 1;
while (ready & i <= predecessors.size()) {
     ClauseActivation predecessor = predecessors.getValue(i-1);
     BooleanValue decisionValue = predecessor.getDecision();

     // Note that the decision will be null if the predecessor clause has not run yet.
     if (decisionValue == null) {
         ready = false;
     } else {
         ready = !decisionValue.value;
     }

     i = i + 1;
}

return ready;
```

[5] receiveControl ( )

```
// If all predecessors to the clause for this activation have run their tests and failed,
then run the test for this clause.
// If the test succeeds, then terminate any other clauses that may be running and run the
body of this clause.
// If the test fails, then pass control to successor clauses.

if (this.isReady()) {
    this.runTest();

    BooleanValue decision = this.getDecision();

    // Note that the decision may be null if the test was terminated before completion.
    if (decision != null) {
        if (decision.value == true) {
            this.selectBody();
        } else {
            ClauseActivationList successors = this.getSuccessors();

            // *** Give control to all successors concurrently. ***
            for (Iterator i = successors.iterator(); i.hasNext();) {
                ClauseActivation successor = (ClauseActivation)i.next();
                successor.receiveControl();
            }
        }
    }
}
```

[6] runTest ( )

```
// Run the test of the clause for this clause activation.

this.conditionalNodeActivation.runTest(this.clause);
```

[7] selectBody ( )

```
// Select the body of the clause for this clause activation.

this.conditionalNodeActivation.selectBody(this.clause);
```

8.10.2.10 ClearAssociationActionActivation

A clear association action activation is an action activation for a clear association action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the extent, at the current execution locus, of the given association.
// Read the object input pin. Destroy all links in which the object participates.

ClearAssociationAction action = (ClearAssociationAction)(this.node);

ExtensionalValueList extent = this.getExecutionLocus().getExtent(action.association);
Value objectValue = this.takeTokens(action.object).getValue(0);

for (int i = 0; i < extent.size(); i++) {
     Link link = (Link)(extent.getValue(i));
     if (this.valueParticipatesInLink(objectValue, link)) {
         link.destroy();
     }
}
```

8.10.2.11 ClearStructuralFeatureActionActivation

A clear structural feature action activation is a structural feature action activation for a clear structural feature action.

Generalizations

- StructuralFeatureActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value of the object input pin.
// If the given feature is an association end, then
// destroy all links that have the object input on the opposite end.
// Otherwise, if the object input is a structured value, then
// set the appropriate feature of the input value to be empty.

ClearStructuralFeatureAction action = (ClearStructuralFeatureAction)(this.node);
StructuralFeature feature = action.structuralFeature;
Association association = this.getAssociation(feature);

Value value = this.takeTokens(action.object).getValue(0);

if (association != null) {
     LinkList links = this.getMatchingLinks(association, feature, value);
     for (int i = 0; i < links.size(); i++) {
          Link link = links.getValue(i);
          link.destroy();
     }
} else if (value instanceof StructuredValue) {

// If the value is a data value, then it must be copied before
     // any change is made.
     if (!(value instanceof Reference)) {
          value = value.copy();
     }

     ((StructuredValue)value).setFeatureValue(action.structuralFeature, new ValueList(), 0);
}

if (action.result != null) {
     this.putToken(action.result, value);
}
```

8.10.2.12 ConditionalNodeActivation

A conditional node activation is a structured activity node activation for a node that is a conditional node.

Generalizations

- StructuredActivityNodeActivation

Attributes

None

Associations

- clauseActivations : ClauseActivation [0..*]
  The activations for each clause in the conditional node for this node activation.
- selectedClause : clause
  The clause chosen from the set of selected clauses to actually be executed.
- selectedClauses : Clause [0..*]
  The set of clauses which meet the conditions to have their bodies activated.

Operations

[1] completeAction ( ) : Token [0..*]

```
// Only complete the conditional node if it is not suspended.

if (!this.isSuspended()) {
     completeBody();
}
return  super.completeAction();
```

[2] completeBody ( )

```
// Complete the activation of the body of a conditional note by
// copying the outputs of the selected clause (if any) to the output
// pins of the node and terminating the activation of all nested nodes.

if (this.selectedClause != null) {
     ConditionalNode node = (ConditionalNode) (this.node);
     OutputPinList resultPins = node.result;
     OutputPin bodyOutputPin = bodyOutputPins.getValue(k);
     this.putTokens(resultPin,  this.getPinValues(bodyOutputPin));
     }
}
this.activationGroup.terminateAll();
```

[3] doStructuredActivity ( )

```
// Run all the non-executable, non-pin nodes in the conditional node.
// Activate all clauses in the conditional node and pass control to those that are ready
(i.e., have no predecessors).
// If one or more clauses have succeeded in being selected, choose one non-deterministically
and run its body, then copy the outputs of that clause to the output pins of the node.
```

```
ConditionalNode node = (ConditionalNode)(this.node);

ActivityNodeActivationList nodeActivations = this.activationGroup.nodeActivations;
ActivityNodeActivationList nonExecutableNodeActivations = new ActivityNodeActivationList();
for (int i = 0; i < nodeActivations.size(); i++) {
     ActivityNodeActivation nodeActivation = nodeActivations.getValue(i);
     if (!(nodeActivation.node instanceof ExecutableNode | nodeActivation.node instanceof
Pin)) {
          nonExecutableNodeActivations.addValue(nodeActivation);
     }
}

this.activationGroup.run(nonExecutableNodeActivations);

this.clauseActivations.clear();
ClauseList clauses = node.clause;
for (int i = 0; i < clauses.size(); i++) {
     Clause clause = clauses.getValue(i);
     ClauseActivation clauseActivation = new ClauseActivation();
     clauseActivation.clause = clause;
     clauseActivation.conditionalNodeActivation = this;
     this.clauseActivations.addValue(clauseActivation);
}

this.selectedClauses.clear();

ClauseActivationList readyClauseActivations = new ClauseActivationList();
for (int i = 0; i < this.clauseActivations.size(); i++) {
     ClauseActivation clauseActivation = this.clauseActivations.getValue(i);
     if (clauseActivation.isReady()) {
          readyClauseActivations.addValue(clauseActivation);
     }
}

// *** Give control to all ready clauses concurrently. ***
for (Iterator i = readyClauseActivations.iterator(); i.hasNext() ;) {
     ClauseActivation clauseActivation = (ClauseActivation)i.next();
     clauseActivation.receiveControl();
}
```

```
this.selectedClause = null;
if (this.selectedClauses.size() > 0 & this.isRunning()) {
     // *** If multiple clauses are selected, choose one non-deterministically. ***
     int i =
((ChoiceStrategy)this.getExecutionLocus().factory.getStrategy("choice")).choose(this.selected
Clauses.size());
     this.selectedClause = this.selectedClauses.getValue(i-1);

     for (int j = 0; j < clauses.size(); j++) {
          Clause clause = clauses.getValue(j);
          if (clause != this.selectedClause) {
               ExecutableNodeList testNodes = clause.test;
               for (int k = 0; k < testNodes.size(); k++) {
                    ExecutableNode testNode = testNodes.getValue(k);
                    this.activationGroup.getNodeActivation(testNode).terminate();
               }
          }
     }

     this.activationGroup.runNodes(this.makeActivityNodeList (this.selectedClause.body));

}
```

```
[4] getClauseActivation ( in clause : Clause ) : ClauseActivation

// Get the clause activation corresponding to the given clause.

ClauseActivation selectedClauseActivation = null;
int i = 1;
while ((selectedClauseActivation == null) & i <= this.clauseActivations.size()) {
     ClauseActivation clauseActivation = this.clauseActivations.getValue(i-1);
     if (clauseActivation.clause == clause) {
          selectedClauseActivation = clauseActivation;
     }
     i = i + 1;
}

return selectedClauseActivation;
```

[5] resume()

```
// When this conditional node is resumed after being suspended, complete
// its body and then resume it as a structured activity node.
// [Note that this presumes that accept event actions are not allowed
// in the test part of a clause of a conditional node.]

completeBody();
super.resume();
```

[6] runTest (in clause : Clause)

```
// Run the test for the given clause.

if (this.isRunning()) {
     this.activationGroup.runNodes(this.makeActivityNodeList(clause.test));
}
```

[7] selectBody (in clause : Clause)

```
// Add the clause to the list of selected clauses.

this.selectedClauses.addValue(clause);
```

8.10.2.13 CreateLinkActionActivation

A create link action activation is a write link action activation for a create link action.

Generalizations

- WriteLinkActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// If the association has any unique ends, then destroy an existing link
// that matches all ends of the link being created.
// Get the extent at the current execution locus of the association for which a link is being
created.
// Destroy all links that have a value for any end for which isReplaceAll is true.
// Create a new link for the association, at the current locus, with the given end data
values,
// inserted at the given insertAt position (for ordered ends).

CreateLinkAction action = (CreateLinkAction)(this.node);
LinkEndCreationDataList endDataList = action.endData;

Association linkAssociation = this.getAssociation();
ExtensionalValueList extent = this.getExecutionLocus().getExtent(linkAssociation);

boolean unique = false;
for (int i = 0; i < endDataList.size(); i++) {
     if (endDataList.getValue(i).end.multiplicityElement.isUnique) {
          unique = true;
     }
}

for (int i = 0; i < extent.size(); i++) {
     ExtensionalValue value = extent.getValue(i);
     Link link = (Link) value;
     boolean match = true;
     boolean destroy = false;
     int j = 1;
     while (j <= endDataList.size()) {
          LinkEndCreationData endData = endDataList.getValue(j - 1);
          if (this.endMatchesEndData(link, endData)) {
               if (endData.isReplaceAll) {
                    destroy = true;
               }
          } else {
               match = false;
          }
          j = j + 1;
     }
     if (destroy | unique & match ) {
          link.destroy();
     }
}

Link newLink = new Link();
newLink.type = linkAssociation;

for (int i = 0; i < endDataList.size(); i++) {
     LinkEndCreationData endData = endDataList.getValue(i);

     int insertAt = 0;
     if (endData.insertAt != null) {
          insertAt = ((UnlimitedNaturalValue) (this
                    .takeTokens(endData.insertAt).getValue(0))).value.naturalValue;
     }

     newLink.setFeatureValue(endData.end,
               this.takeTokens(endData.value), insertAt);
}

newLink.addTo(this.getExecutionLocus());
```

8.10.2.14 CreateObjectActionActivation

A create object action activation is an action activation for a create object action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Create an object with the given classifier (which must be a class) as its type, at the
same locus as the action activation.
// Place a reference to the object on the result pin of the action.

CreateObjectAction action = (CreateObjectAction)(this.node);

Reference reference = new Reference();
reference.referent = this.getExecutionLocus().instantiate((Class_)(action.classifier));

this.putToken(action.result, reference);
```

8.10.2.15 DestroyLinkActionActivation

A destroy link action activation is a write link action activation for a destroy link action.

Generalizations

- WriteLinkActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the extent, at the current execution locus, of the association for which links are
being destroyed.
// Destroy all links that match the given link end destruction data.
// For unique ends, or non-unique ends for which isDestroyDuplicates is true, match links
with a matching value for that end.
// For non-unique, ordered ends for which isDestroyDuplicates is false, match links with an
end value at the given destroyAt position. [Must a value be given, too, in this case?]
// For non-unique, non-ordered ends for which isDestroyDuplicates is false, pick one matching
link (if any) non-deterministically. [The semantics of this case is not clear from the
current spec.]


DestroyLinkAction action = (DestroyLinkAction)(this.node);
LinkEndDestructionDataList destructionDataList = action.endData;


boolean destroyOnlyOne = false;
int j = 1;
while (!destroyOnlyOne & j <= destructionDataList.size()) {
     LinkEndDestructionData endData = destructionDataList.getValue(j-1);
     destroyOnlyOne = !endData.end.multiplicityElement.isUnique & !
endData.end.multiplicityElement.isOrdered  &  !endData.isDestroyDuplicates;
    j = j + 1;
}


LinkEndDataList endDataList = new LinkEndDataList();
for (int i = 0; i < destructionDataList.size(); i++) {
     LinkEndDestructionData endData = destructionDataList.getValue(i);
     endDataList.addValue(endData);
}
```

```
ExtensionalValueList  extent  =  this.getExecutionLocus().getExtent(this.getAssociation());
ExtensionalValueList matchingLinks = new ExtensionalValueList();

for (int i = 0; i < extent.size(); i++) {
     ExtensionalValue value = extent.getValue(i);
     Link link = (Link)value;
     if (this.linkMatchesEndData(link, endDataList)) {
          matchingLinks.addValue(link);
     }
}

// Now that matching is done, ensure that all tokens on end data input pins
// are consumed.
for (int i = 0; i < destructionDataList.size(); i++) {
     LinkEndDestructionData endData = destructionDataList.getValue(i);
     Property end = endData.end;
     if (!endData.isDestroyDuplicates
          & !end.multiplicityElement.isUnique & end.multiplicityElement.isOrdered) {
          this.takeTokens(endData.destroyAt);
     }
     this.takeTokens(endData.value);
}

if (destroyOnlyOne) {
     // *** If there is more than one matching link, non-deterministically choose one. ***
     if (matchingLinks.size() > 0) {
          int i =
((ChoiceStrategy)this.getExecutionLocus().factory.getStrategy("choice")).choose(matchingLinks
.size());
          matchingLinks.getValue(i-1).destroy();
     }
} else {
     for (int i = 0; i < matchingLinks.size(); i++) {
          ExtensionalValue matchingLink = matchingLinks.getValue(i);
          matchingLink.destroy();
     }
}
```

8.10.2.16 DestroyObjectActionActivation

A destroy object action activation is an action activation for a destroy object action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] destroyObject (in value : Value, in isDestroyLinks : Boolean, in isDestroyOwnedObjects : Boolean)

```
// If the given value is a reference, then destroy the referenced object, per the given
destroy action attribute values.

if (value instanceof Reference) {
     Reference reference = (Reference)value;

     if (isDestroyLinks | isDestroyOwnedObjects) {
          ExtensionalValueList extensionalValues = this.getExecutionLocus().extensionalValues;
          for (int i = 0; i < extensionalValues.size(); i++) {
               ExtensionalValue extensionalValue = extensionalValues.getValue(i);
               if (extensionalValue instanceof Link) {
                    Link link = (Link)extensionalValue;
                    if (this.valueParticipatesInLink(reference, link)) {
                        if (isDestroyOwnedObjects) {
                            Value compositeValue =
                                 this.getCompositeValue(reference, link);
                            if (compositeValue != null) {
                                 this.destroyObject(compositeValue, isDestroyLinks,
                                      isDestroyOwnedObjects);
                            }
                        }
                        if (isDestroyLinks & !link.getTypes().isEmpty()) {
                            link.destroy();
                        }
                    }
               }
          }
     }

     if (isDestroyOwnedObjects) {
         FeatureValueList objectFeatureValues = reference.getFeatureValues();
         for (int i = 0; i < objectFeatureValues.size(); i++) {
             FeatureValue featureValue = objectFeatureValues.getValue(i);
             if (((Property)featureValue.feature).aggregation == AggregationKind.composite) {
                 ValueList values = featureValue.values;
                 for (int j = 0; j < values.size(); j++) {
                     Value ownedValue = values.getValue(j);
                     this.destroyObject(ownedValue, isDestroyLinks, isDestroyOwnedObjects);
                 }
             }
         }
     }

     reference.destroy();
}
```

[2] doAction ( )

```
// Get the value on the target input pin.
// If the value is not a reference, then the action has no effect.
// Otherwise, do the following.
// If isDestroyLinks is true, destroy all links in which the referent participates.
// If isDestroyOwnedObjects is true, destroy all objects owned by the referent via
// either composite attributes or composition links.
// Destroy the referent object.

DestroyObjectAction action = (DestroyObjectAction)(this.node);
Value value = this.takeTokens(action.target).getValue(0);

this.destroyObject(value, action.isDestroyLinks, action.isDestroyOwnedObjects);
```

[3] getCompositeValue (in reference : Reference, in link : Link) : Value [0..1]

```
// If the given reference participates in the given link as a composite,
// then return the opposite value. Otherwise return null.

FeatureValueList linkFeatureValues = link.getFeatureValues();

Value compositeValue = null;
int i = 1;
while (compositeValue == null & i <= linkFeatureValues.size()) {
    FeatureValue featureValue = linkFeatureValues.getValue(i - 1);
    Value value = featureValue.values.getValue(0);
    if (!value.equals(reference) &
        ((Property) featureValue.feature).aggregation == AggregationKind.composite) {
        compositeValue = value;
    }
    i = i + 1;
}

return compositeValue;
```

8.10.2.17 ExpansionActivationGroup

An expansion activation group is an activity node activation group used for activating nodes inside an expansion region.

It functions just like a normal activation group, except it has output pin activations corresponding to the input pins and the expansion nodes of the expansion region.

Instances of edges from nodes inside the expansion region that connect to region input pins, input expansion nodes or output expansion nodes are redirected to connect to the corresponding "region input," "group input," or "group output" pin, respectively.

Generalizations

- ActivityNodeActivationGroup

Attributes

- index : Integer
  The index (starting at 1) of this activation group in the list held by the expansion region activation.

Associations

- groupInputs : OutputPinActivation [1..*]
  Output pin activations corresponding, in order, to the input expansion nodes of the expansion region of this activation group.
- groupOutputs : OutputPinActivation [0..*]
  Output pin activations corresponding, in order, to the output expansion nodes of the expansion region of this activation group.
- regionActivation : ExpansionRegionActivation
  The expansion region activation this activation group is for.
- regionInputs : OutputPinActivation [0..*]
  Output pin activations corresponding, in order, to the input pins of the expansion region of this activation group.

Operations

[1] getActivityExecution ( ) : ActivityExecution

```
// Get the activity execution that contains the expansion region activation for this
activation group.
```

```
return  this.regionActivation.getActivityExecution();
```

[2] getNodeActivation (in node : ActivityNode) : ActivityNodeActivation

```
// If the given node is an input pin of the expansion region, then return the corresponding
region-input output-pin activation.
// If the given node is an input expansion node of the expansion region, then return the
corresponding group-input output-pin activation.
// If the given node is an output expansion node of the expansion region, then return the
corresponding group-output output-pin activation.
// Otherwise return the node activation from the activation group, as usual.

ExpansionRegion region = (ExpansionRegion)(this.regionActivation.node);

InputPinList inputs = region.input;
ActivityNodeActivation activation = null;

int i = 1;
while (activation == null & i <= region.input.size()) {
     if (node == region.input.getValue(i-1)) {
          activation = this.regionInputs.getValue(i-1);
     }
     i = i + 1;
}

int j = 1;
while (activation == null & j <= region.inputElement.size()) {
     if (node == region.inputElement.getValue(j - 1)) {
          activation = this.groupInputs.getValue(j - 1);
     }
     j = j + 1;
}

int k = 1;
while (activation == null & k <= region.outputElement.size()) {
     if (node == region.outputElement.getValue(k - 1)) {
          activation = this.groupOutputs.getValue(k - 1);
     }
     k = k + 1;
}
```

```
if (activation == null) {
     activation = super.getNodeActivation(node);
}

return activation;
```

[2] resume (in activation : ActivityNodeActivation)

```
// Resume the given activation in this activation group. If this is the
// last suspended activation, then resume the associated region
// activation.

super.resumt(activation);
if (!this.isSuspended()) {
     this.regionActivation.resume(this);
}
```

[3] suspend (in activation : ActivityNodeActivation)

```
// Suspend the given activation in this activation group. If this is
// the only suspended activation, then suspend the associated region
// activation.

if (!this.issuspended()) {
     this.regionActivation.suspend();
}
super.suspend(activation);
```

8.10.2.18 ExpansionNodeActivation

Generalizations

- ObjectNodeActivation

Attributes

None

Associations

None

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// Take tokens from all incoming edges.
```

```
this.addTokens(incomingTokens);
```

[2] getExpansionRegionActivation ( ) : ExpansionRegionActivation

```
// Return the expansion region activation corresponding to this expansion node, in the
context of the activity node activation group this expansion node activation is in.

ExpansionNode node = (ExpansionNode)(this.node);

ExpansionRegion region = node.regionAsInput;
if (region == null) {
    region = node.regionAsOutput;
}

return  (ExpansionRegionActivation)(this.group.getNodeActivation(region));
```

[3] isReady ( ) : Boolean

```
// An expansion node is always fired by its expansion region.

return false;
```

[4] receiveOffer ( )

```
// Forward the offer on to the expansion region.

this.getExpansionRegionActivation().receiveOffer();
```

8.10.2.19 ExpansionRegionActivation

An expansion region activation is an action activation for a node that is an expansion region.

Note that even though an expansion region is a structured activity node, an expansion region activation is not a structured activity activation because of the special nature of expansion region behavior.

Generalizations

- ActionActivation

Attributes

- next : Integer[0..1]
  The index of the next activation group to be run, if the expansion region is iterative.

Associations

- activationGroups : ExpansionActivationGroup [0..*]
  The set of expansion activation groups for this expansion region activation. One activation group is created corresponding to each token held by the first input expansion node activation for the expansion region.
- inputExpansionTokens : TokenSet [1..*]
  The tokens taken from each of the input expansion node activations for this expansion region activation. These are preserved for initializing the group input of each of the activation groups.
- inputTokens : TokenSet [0..*]
  The tokens taken from each of the input pin activations for this expansion region activation. These are preserved for initializing the region inputs of each of the activation groups.

Operations

[1] doAction ( )

```
// If the expansion region has mustIsolate=true, then carry out its behavior with isolation.
// Otherwise just activate it normally.

if (((StructuredActivityNode)(this.node)).mustIsolate) {
     _beginIsolation();
          this.doStructuredActivity();
     _endIsolation();
} else {
     this.doStructuredActivity();
}
```

[2] doOutput ( )

```
ExpansionRegion region = (ExpansionRegion) this.node;
ExpansionNodeList outputElements = region.outputElement;

if (!this.isSuspended()) {
     for (int i = 0; i < activationGroups.size(); i++) {
          ExpansionActivationGroup activationGroup = activationGroups.getValue(i);
          OutputPinActivationList groupOutputs = activationGroup.groupOutputs;
          for (int j = 0; j < groupOutputs.size(); j++) {
               OutputPinActivation groupOutput = groupOutputs.getValue(j);
               ExpansionNode outputElement = outputElements.getValue(j);
               this.getExpansionNodeActivation(outputElement).addTokens(
                         groupOutput.takeTokens());
          }
     }
}
```

[3] doStructuredActivity ( )

```
// Create a number of expansion region activation groups equal to the number of values
expanded in the region,
// setting the region inputs and group inputs for each group.
// Run the body of the region in each group, either iteratively or in parallel.
// Add the outputs of each activation group to the corresponding output expansion node
activations.

ExpansionRegion region = (ExpansionRegion)this.node;
InputPinList inputPins = region.input;
ExpansionNodeList inputElements = region.inputElement;
ExpansionNodeList outputElements = region.outputElement;

this.activationGroups.clear();
int n = this.inputExpansionTokens.getValue(0).tokens.size();
int k = 1;
while (k <= n) {
    ExpansionActivationGroup activationGroup = new ExpansionActivationGroup();
    activationGroup.regionActivation = this;
    activationGroup.index = k;

    int j = 1;
    while (j <= inputPins.size()) {
        OutputPinActivation regionInput = new OutputPinActivation();
        regionInput.run();
        activationGroup.regionInputs.addValue(regionInput);
        j = j + 1;
    }

    j = 1;
    while (j <= inputElements.size()) {
        OutputPinActivation groupInput = new OutputPinActivation();
        groupInput.run();
        activationGroup.groupInputs.addValue(groupInput);
        j = j + 1;
    }

    j = 1;
    while (j <= outputElements.size()) {
        OutputPinActivation groupOutput = new OutputPinActivation();
        groupOutput.run();
```

```
          activationGroup.groupOutputs.addValue(groupOutput);
         j = j + 1;
     }

      activationGroup.createNodeActivations(region.node);
      activationGroup.createEdgeInstances(region.edge);
      this.activationGroups.addValue(activationGroup);

     k = k + 1;
}

ExpansionActivationGroupList activationGroups = this.activationGroups;

if (region.mode == ExpansionKind.iterative) {
     this.next = 1;
     this.runIterative();
}
else if (region.mode == ExpansionKind.parallel) {
     this.runParallel();
}

this.doOutput();
```

[4] getExpansionNodeActivation (in node : ExpansionNode) : ExpansionNodeActivation

```
// Return the expansion node activation corresponding to the given expansion node, in the context of the activity node activation group this expansion region activation is in.
// [Note: Expansion regions do not own their expansion nodes. Instead, they are own as object nodes by the enclosing activity or group.
// Therefore, they will already be activated along with their expansion region.]

return (ExpansionNodeActivation)(this.group.getNodeActivation(node));
```

[5] isSuspended() : Boolean

```
// Check if the activation group for this node is suspended.

boolean suspended = false;

int i = 1;
while (i <= this.activationGroups.size() & !suspended) {
     ActivityNodeActivationGroup group = this.activationGroups.get(i-1);
     suspended = group.isSuspended();
     i = i + 1;
}

return suspended;
```

[6] numberOfValues ( ) : Integer

```
// Return the number of values to be acted on by the expansion region of
// this activation, which is the minimum of the number of values offered
// to each of the input expansion nodes of the activation.

ExpansionRegion region = (ExpansionRegion) (this.node);
ExpansionNodeList inputElements = region.inputElement;

int n = this.getExpansionNodeActivation(inputElements.getValue(0))
                   .countOfferedValues();
int i = 2;
while (i <= inputElements.size()) {
     int count = this.getExpansionNodeActivation(
               inputElements.getValue(i - 1)).countOfferedValues();
     if (count < n) {
          n = count;
     }
     i = i + 1;
}
return n;
```

[7] resume (in activationGroup : ExpansionActivationGroup)

```
// Resume an expansion region after the suspension of the given
// activation group. If the region is iterative, then continue with the
// iteration. If the region is parallel, and there are no more suspended
// activation groups, then generate the expansion node output.

ExpansionRegion region = (ExpansionRegion) this.node;

this.resume();
this.terminateGroup(activationGroup);
if (region.mode == ExpansionKind.iterative) {
     this.runIterative();
}

this.doOutput();
```

[8] runGroup (in activationGroup : ExpansionActivationGroup)

```
// Set up the inputs for the group with the given index, run the group and then fire the
group outputs.

if (this.isRunning()) {
     TokenSetList inputTokens = this.inputTokens;
     for (int j = 0; j < inputTokens.size(); j++) {
          TokenSet tokenSet = inputTokens.getValue(j);
          OutputPinActivation regionInput = activationGroup.regionInputs.getValue(j);
          regionInput.clearTokens();
          regionInput.addTokens(tokenSet.tokens);
          regionInput.sendUnofferedTokens();
     }

     TokenSetList inputExpansionTokens = this.inputExpansionTokens;
     for (int j = 0; j < inputExpansionTokens.size(); j++) {
          TokenSet tokenSet = inputExpansionTokens.getValue(j);
          OutputPinActivation groupInput = activationGroup.groupInputs.getValue(j);
          groupInput.clearTokens();
          if (tokenSet.tokens.size() >= activationGroup.index) {
               groupInput.addToken(tokenSet.tokens.getValue(activationGroup.index - 1));
          }
          groupInput.sendUnofferedTokens();
     }

     activationGroup.run(activationGroup.nodeActivations);

     this.terminateGroup(activationGroup);
}
```

[9] runIterative ( )

```
// Run the body of the region iteratively, either until all activation
// groups have run or until the region is suspended.

ExpansionActivationGroupList activationGroups = this.activationGroups;
```

```
while (this.next <= activationGroups.size() & !this.isSuspended()) {
     ExpansionActivationGroup activationGroup = activationGroups.getValue(this.next-1);
     this.runGroup(activationGroup);
     this.next = this.next + 1;
}
```

[10] runParallel ( )

```
// Run the body of the region concurrently.

ExpansionActivationGroupList activationGroups = this.activationGroups;

// *** Activate all groups concurrently. ***
for (Iterator i = activationGroups.iterator(); i.hasNext();) {
     ExpansionActivationGroup activationGroup = (ExpansionActivationGroup) i.next();
     this.runGroup(activationGroup);
}
```

[11] sendOffers ( )

```
// Fire all output expansion nodes and send offers on all outgoing control flows.

ExpansionRegion region = (ExpansionRegion)(this.node);

// *** Send offers from all output expansion nodes concurrently. ***
ExpansionNodeList outputElements = region.outputElement;
for (Iterator i = outputElements.iterator(); i.hasNext();) {
     ExpansionNode outputElement = (ExpansionNode)i.next();
      this.getExpansionNodeActivation(outputElement).sendUnofferedTokens();
}

// Send offers on all outgoing control flows.
super.sendOffers();
```

[12] takeOfferedTokens ( ) : Token [0..*]

```
// Take the tokens from the input pin and input expansion node activations and save them.

super.takeOfferedTokens();

ExpansionRegion region = (ExpansionRegion)(this.node);
InputPinList inputPins = region.input;
ExpansionNodeList inputElements = region.inputElement;
```

```
this.inputTokens.clear();
this.inputExpansionTokens.clear();

for (int i = 0; i < inputPins.size(); i++) {
     InputPin inputPin = inputPins.getValue(i);
     TokenSet tokenSet = new TokenSet();
     tokenSet.tokens = this.getPinActivation(inputPin).takeTokens();
     this.inputTokens.addValue(tokenSet);
}

int n = this.numberOfValues();
for (int i = 0; i < inputElements.size(); i++) {
     ExpansionNode inputElement = inputElements.getValue(i);
     ExpansionNodeActivation expansionNodeActivation =
this.getExpansionNodeActivation(inputElement);
     expansionNodeActivation.fire(expansionNodeActivation.takeOfferedTokens());
     TokenList tokens = expansionNodeActivation.takeTokens();
     TokenSet tokenSet = new TokenSet();
     int j = 1;
     while (j <= n) {
          tokenSet.tokens.add(tokens.getValue(j-1));
          j = j + 1;
     }
     this.inputExpansionTokens.addValue(tokenSet)
}

return new TokenList();
```

[13] terminate ( )

```
// Terminate the execution of all contained node activations (which completes the performance
of the expansion region activation).

ExpansionActivationGroupList activationGroups = this.activationGroups;
for (int i = 0; i < activationGroups.size(); i++) {
     ExpansionActivationGroup activationGroup = this.activationGroups.getValue(i);
     OutputPinActivationList groupOutputs = activationGroup.groupOutputs;

     _beginIsolation();
     for (int j = 0; j < groupOutputs.size(); j++) {
          OutputPinActivation groupOutput = groupOutputs.getValue(j);
          groupOutput.fire(groupOutput.takeOfferedTokens());
     }
     activationGroup.terminateAll();
     _endIsolation();
}

super.terminate();
```

[14] terminateGroup (in activationGroup : ExpansionActivationGroup)

```
if (this.isRunning() & !this.isSuspended()) {
     OutputPinActivationList groupOutputs = activationGroup.groupOutputs;
     for (int i = 0; i < groupOutputs.size(); i++) {
          OutputPinActivation groupOutput = groupOutputs.getValue(i);
          groupOutput.fire(groupOutput.takeOfferedTokens());
     }

     activationGroup.terminateAll();
}
```

8.10.2.20 InputPinActivation

An input pin activation is a pin activation for an input pin.

Generalizations

- PinActivation

Attributes

None

Associations

- streamingParameterValue : StreamingParameterValue [0..1]
  The streaming parameter value to which values from tokens accepted by this input pin activation should be posted, if the input pin for this activation corresponds to a streaming parameter of an invoked behavior.

Operations

`[1] fire ( incomingTokens : Token [0..*] )`

// Add all incoming tokens to the pin.

// If the pin activation is streaming, and there are incoming tokens,

// then post the values from the tokens to the streaming parameter value.

// Then check if the streaming parameter value has terminated and, if so,

```
// terminate the action activation.

super.fire(incomingTokens);

if (this.isStreaming() & incomingTokens.size() > 0) {
   ValueList values = new ValueList();
   for (int i = 0; i < incomingTokens.size(); i++) {
      Token token = incomingTokens.getValue(i);
      Value value = token.getValue();
      if (value != null) {
         values.addValue(value);
      }
   }
   this.streamingParameterValue.post(values);

   if (this.streamingIsTerminated()) {
      if (this.actionActivation instanceof CallActionActivation) {
         ((CallActionActivation)this.actionActivation).completeStreamingCall();
      }
   }
}
```

[2] getTotalValueCount ( ) : Integer

```
// Return the total number of values already being offered by the
// pin plus those being offered by the sources of incoming edges.

return this.countUnofferedTokens() + this.countOfferedValues();
```

[3] isReady ( ) : Boolean

```
// If this pin activation is not streaming, then return true if the total
// number of values already being offered by the pin plus those being
// offered by the sources of incoming edges is at least equal to the
// minimum multiplicity of the pin.
// If this pin activation is streaming, then return true if the minimum
// multiplicity is zero or if there is at least one offered value.

boolean ready = super.isReady();
if (ready) {
    int minimum = ((Pin) this.node).multiplicityElement.lower;
    if (this.isStreaming()) {
        if (minimum > 0) {
            minimum = 1;
        }
    }
    ready = this.getTotalValueCount() >= minimum;
}

return ready;
```

[4] isReadyForStreaming ( ) : Boolean

```
// Return true if this pin activation is ready assuming that it
// corresponds to a streaming parameter. In this case, it is
// ready if it has a lower multiplicity bound of zero, or if
// there is at least one offered value.

return super.isReady() &
        (((Pin) this.node).multiplicityElement.lower == 0 |
         getTotalValueCount() >= 1);
```

[5] isStreaming ( ) : Boolean

```
// Return true if this pin activation is for a pin that corresponds
// to a streaming input parameter.

return this.streamingParameterValue != null;
```

[6] receiveOffer ( )

```
// If this pin activation is streaming, then accept offered tokens
// up to the multiplicity upper bound of the pin and fire on the
// accepted tokens.
// If the pin activation is not streaming, then forward the offer
// to the action activation. (When all input pins are ready, the
// action will fire them.)

if (this.isStreaming()) {
    super.receiveOffer();
} else {
    this.actionActivation.receiveOffer();
}
```

[7] streamingIsTerminated ( ) : Boolean

```
boolean isTerminated = false;

_beginIsolation();
isTerminated = this.streamingParameterValue.isTerminated();
 _endIsolation();

return isTerminated;
```

8.10.2.21 InvocationActionActivation

An invocation action activation is an action activation of an invocation action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

None

8.10.2.22 LinkActionActivation

A link action activation is an action activation for a link action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] endMatchesEndData (in link : Link, in endData : LinkEndData) : Boolean

```
// Test whether the appropriate end of the given link matches the given end data.

boolean matches = false;
if (endData.value == null) {
    matches = true;
} else {
    Property end = endData.end;
    FeatureValue linkFeatureValue = link.getFeatureValue(end);
    Value endValue = this.getTokens(endData.value).getValue(0);
    if (endData instanceof LinkEndDestructionData) {
            if (!((LinkEndDestructionData)endData).isDestroyDuplicates & !
end.multiplicityElement.isUnique & end.multiplicityElement.isOrdered)  {
                int destroyAt = ((UnlimitedNaturalValue)
(this.getTokens(((LinkEndDestructionData)endData).destroyAt).getValue(0))).value.naturalValue
;
                matches = linkFeatureValue.values.getValue(0).equals(endValue) &&
linkFeatureValue.position == destroyAt;
            } else {
                matches = linkFeatureValue.values.getValue(0).equals(endValue);
            }
    } else {
        matches =  linkFeatureValue.values.getValue(0).equals(endValue);
    }
}

return matches;
```

[2] getAssociation ( ) : Association

```
// Get the association for the link action of this activation.

return  (Association)(((LinkAction)(this.node)).endData.getValue(0).end.association);
```

[3] linkMatchesEndData (in link : Link, in endDataList : LinkEndData [0..*]) : Boolean

```
// Test whether the given link matches the given end data.

boolean matches = true;
int i = 1;
while (matches & i <= endDataList.size()) {
     matches = this.endMatchesEndData(link, endDataList.getValue(i-1));
    i = i + 1;
}

return matches;
```

8.10.2.23 LoopNodeActivation

A loop node activation is a structured activity node activation for a node that is a loop node.

Generalizations

- StructuredActivityNodeActivation

Attributes

- isTerminateAll : Boolean

Associations

- bodyOutputLists : Values [0..*]

Operations

[1] continueLoop ( )

```
// Continue the loop node when it is resumed after being suspended. If
// isTestedFirst is true, then continue executing the loop. If
// isTestedFirst is false, then run the test to determine whether
// the loop should be continued or completed.
// [Note that this presumes that an accept event action is not allowed
// in the test part of a loop node.]

LoopNode loopNode = (LoopNode) (this.node);
boolean continuing = true;
if (!loopNode.isTestedFirst) {
     continuing = this.runTest();
}
if (this.isRunning()) {
     this.activationGroup.terminateAll();
     this.doLoop(continuing);
}
```

[2] createNodeActivations ( )

```
// In addition to creating activations for contained nodes, create activations for any loop
variables.

super.createNodeActivations();
this.activationGroup.createNodeActivations(this.makeLoopVariableList());
```

[3] doLoop (in continuing : Boolean)

```
// If isTestedFirst is true, then repeatedly run the test part and the
// body part of the loop, copying values from the body outputs to the
//loop variables.
// If isTestedFirst is false, then repeatedly run the body part and the
// test part of the loop, copying values from the body outputs to the
// loop variables.

LoopNode loopNode = (LoopNode) (this.node(;
OutputPinList loopVariables = loopNode.loopVariable;
OutputPinList resultPins = loopNode.result;

while (continuing) {

     // Set loop variable values
     this.runLoopVariables();
     for (int i = 0; i < loopVariables.size() i++) {
          OutputPin loopVariable = loopVariables.getValue(i);
          Values bodyOutputList = bodyOutputLists.getValue(i);
          ValueList values = bodyOutputList.values;
          this.putPinValues(loopVariable, values);
          ((OutputPinActivation) this.activationGroup
                    .getNodeActivation(loopVariable)).sendUnofferedTokens();
     }

     // Run all the non-executable, non-pin nodes in the conditional
     // node.
     ActivityNodeActivationList nodeActivations = this.activationGroup.nodeActivations;
     ActivityNodeActivationList nonExecutableNodeActivations =
          new ActivityNodeActivationList();
     for (int i = 0; i < nodeActivations.size(); i++) {
          ActivityNodeActivation nodeActivation = nodeActivations
                    .getValue(i);
          if (!nodeActivation.node instanceof ExecutableNode |
               nodeActivation.node instanceof Pin)) {
               nonExecutableNodeActivations.addValue(nodeActivation);
          }
     }
     this.activationGroup.run(nonExecutableNodeActivations);

     // Run the loop
     if (loopNode.isTestedFirst) {
          continuing = this.runTest();
          if (continuing) {
               this.runBody();
          {
     } else {
          this.runBody();
          if (this.isRunning() & !this.isSuspended()) {
               continuing = this.runTest();
          }
     }

     if (this.isTerminateAll & this.isRunning() & !this.isSuspended()) {
               this.activationGroup.terminateAll();
     } else {
          continuing = false;
     }

}

if (!this.isTerminateAll & this.isRunning() & !this.isSuspended()) {
     for (int i = 0; i < bodyOutputLists.size(); i++) {
          Values bodyOutputList = bodyOutputLists.getValue(i);
          OutputPin resultPin = resultPins.getValue(i);
          this.putTokens(resultPin, bodyOutputList.values);
     }
}
```

[4] doStructuredActivity ( )

```
// Set the initial values for the body outputs to the values of the loop variable input pins.
// If isTestedFirst is true, then repeatedly run the test part and the body part of the loop,
copying values from the body outputs to the loop variables.
// If isTestedFirst is false, then repeatedly run the body part and the test part of the
loop, copying values from the body outputs to the loop variables.
// When the test fails, copy the values of the body outputs to the loop outputs.
// [Note: The body outputs are used for the loop outputs, rather than the loop variables,
since values on the loop variables may be consumed when running the test for the last time.]

LoopNode loopNode = (LoopNode)(this.node);
InputPinList loopVariableInputs = loopNode.loopVariableInput;
this.bodyOutputLists.clear();
for (int i = 0; i < loopVariableInputs.size(); i++) {
     InputPin loopVariableInput = loopVariableInputs.getValue(i);
     Values bodyOutputList = new Values();
     bodyOutputList.values = this.takeTokens(loopVariableInput);
     this.bodyOutputLists.addValue(bodyOutputList);
}

this.isTerminate = false;
this.doLoop(true);
```

[5] makeLoopVariableList ( ) : ActivityNode [0..*]

```
// Return an activity node list containing the loop variable pins for the loop node of this
activation.

LoopNode loopNode = (LoopNode)(this.node);
ActivityNodeList nodes = new ActivityNodeList();

OutputPinList loopVariables = loopNode.loopVariable;
for (int i = 0; i <  loopVariables.size(); i++) {
     OutputPin loopVariable = loopVariables.getValue(i);
     nodes.addValue(loopVariable);
}

return nodes;
```

[6] runBody ( )

```
// Run the body part of the loop node for this node activation and save the body outputs.

LoopNode loopNode = (LoopNode)this.node;

this.activationGroup.runNodes(this.makeActivityNodeList(loopNode.bodyPart));
```

```
if (!this.terminateAll & !this.isSuspended()) {
     this.saveBodyOutputs();
}
```

[7] resume ( )

```
// When this loop node is resumed after being suspended, continue with
// its next iteration (is any). Once the loop has completed execution
// without being suspended again, complete the action.

LoopNode loopNode = (LoopNode) (this.node);

this.saveBodyOutputs();

if (!this.isTerminateAll) {
     if (loopNode.mustIsolate) {
            beginIsolation();
          this.continueLoop();
            endIsolation();
     } else {
          this.continueLoop();
     }
}

if (this.isSuspended()) {

     // NOTE: If the subsequent iteration of the loop suspends it again,
     // then it is necessary to remove the previous suspension from the
     // containing activity node activation group.
     this.group.resume(this);
} else {
     super.resume();
}
```

[8] runLoopVariables ( )

```
// Run the loop variable pins of the loop node for this node activation.

this.activationGroup.runNodes(this.makeLoopVariableList());
```

[9] runTest ( ) : Boolean

```
// Run the test part of the loop node for this node activation.
// Return the value on the decider pin.

LoopNode loopNode = (LoopNode)(this.node);

this.activationGroup.runNodes(this.makeActivityNodeList(loopNode.test));

ValueList values = this.getPinValues(loopNode.decider);

// If there is no decider value, treat it as false.
boolean decision = false;
if (values.size() > 0) {
   decision = ((BooleanValue)(values.getValue(0))).value;
}

return decision;
```

[10] saveBodyOutputs ( )

```
// Save the body outputs for use in the next iteration.

LoopNode loopNode = (LoopNode) this.node;
OutputPinList bodyOutputs = loopNode.bodyOutput;
ValuesList bodyOutputLists = this.bodyOutputLists;
for (int i = 0; i < bodyOutputs.size(); i++ {
     OutputPin bodyOutput = bodyOutputs.getValue(i);
     Values bodyOutputList = bodyOutputLists.getValue(i);
     bodyOutputList.values = this.getPinValues(bodyOutput);
}
```

[11] terminateAll ( )

```
// Copy the values of the body outputs to the loop outputs, and then
// terminate all activations in the loop.

this.isTerminateAll = true;

LoopNode loopNode = (LoopNode) this.node;
OutputPinList bodyOutputs = loopNode.bodyOutput;
OutputPinList resultPins = loopNode.result;
for (int i = 0; i < bodyOutputs.size(); i++) {
     OutputPin bodyOutput = bodyOutputs.getValue(i);
```

```
    OutputPin resultPin = resultPins.getValue(i);
    this.putTokens(resultPin, this.getPinValues(bodyOutput));
}

super.terminateAll();
```

8.10.2.24 OutputPinActivation

An output pin activation is a pin activation for an output pin.

Generalizations

- PinActivation

Attributes

None

Associations

None

Operations

None

8.10.2.25 PinActivation

A pin activation is an object node activation for a node that is a pin.

Generalizations

- ObjectNodeActivation

Attributes

None

Associations

- actionActivation : ActionActivation [0..1]
  The activation of the action that owns the pin for this pin activation.

Operations

[1] fire (in incomingTokens : Token [0..*])

```
// Add all incoming tokens to the pin.

this.addTokens(incomingTokens);
```

[2] takeOfferedTokens ( ) : Token [0..*]

```
// Take only a number of tokens only up to the limit allowed by
// the multiplicity upper bound of the pin for this activation.
```

```
int count = this.countUnofferedTokens();
int upper = -1;

// Note: A pin activation used in an expansion activation group
// will have this.node == null.
if (this.node != null) {
    upper = ((Pin)(this.node)).multiplicityElement.upper.naturalValue;
}

TokenList tokens = new TokenList();

// Note: upper < 0 indicates an unbounded upper multiplicity.
if (upper < 0 | count < upper) {
    ActivityEdgeInstanceList incomingEdges = this.incomingEdges;
    for (int i=0; i<incomingEdges.size(); i++) {
        ActivityEdgeInstance edge = incomingEdges.getValue(i);
        int incomingCount = edge.countOfferedValues();
        TokenList incomingTokens = new TokenList();
        if (upper < 0 | incomingCount < upper - count) {
            incomingTokens = edge.takeOfferedTokens();
            count = count + incomingCount;
        } else if (count < upper) {
            incomingTokens = edge.takeOfferedTokens(upper-count);
            count = upper;
        }
        for (int j = 0; j < incomingTokens.size(); j++) {
            Token token = incomingTokens.getValue(j);
            tokens.addValue(token);
        }
    }
}

return tokens;
```

8.10.2.26 PinStreamingParameterListener

A pin streaming parameter listener is a streaming parameter listener for posting values from a streaming parameter value to a pin (which should normally be an output pin).

Generalizations

- StreamingParameterListener

Attributes

None

Associations

- nodeActivation : PinActivation
  The node activation for the pin to which streaming parameter values are to be posted.

Operations

[1] isTerminated ( ) : Boolean

```
// This listener is terminated if the node activation is not running.

return !this.nodeActivation.isRunning();
```

[2] post (values : Value [0..*])

```
// Fire the pin activation passing the posted values as incoming tokens,
// then have the pin activation immediately offer these tokens (since
// the pin activation would otherwise not offer them until its
// associated action activation terminates).

TokenList tokens = new TokenList();
for (int i = 0; i < values.size(); i++) {
        Value value = values.getValue(i);
        ObjectToken token = new ObjectToken();
        token.value = value;
        tokens.addValue(token);
}

nodeActivation.fire(tokens);
nodeActivation.sendUnofferedTokens();
```

8.10.2.27 RaiseExceptionActionActivation

A raise exception action activation is an action activation for a raise exception action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value on the exception pin and propagate it as an exception.

RaiseExceptionAction action = (RaiseExceptionAction)this.node;
Value exception = this.takeTokens(action.exception).getValue(0);

this.propagateException(exception);
```

8.10.2.28 ReadExtentActionActivation

A read extent action activation is an action activation for a read extent action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the extent, at the current execution locus, of the classifier (which must be a class) 
identified in the action.
// Place references to the resulting set of objects on the result pin.

ReadExtentAction action = (ReadExtentAction)(this.node);
ExtensionalValueList objects = this.getExecutionLocus().getExtent(action.classifier);

ValueList references = new ValueList();
for (int i = 0; i < objects.size(); i++) {
     Value object = objects.getValue(i);
     Reference reference = new Reference();
     reference.referent = (Object_)object;
     references.addValue(reference);
}

this.putTokens(action.result, references);
```

8.10.2.29 ReadIsClassifiedObjectActionActivation

A read-is-classified object activation is an action activation for a read-is-classified object action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value on the object input pin and determine if it is classified by the classifier
specified in the action.
// If the isDirect attribute of the action is false, then place true on the result output pin
if the input object has the specified classifier or of one its (direct or indirect)
descendants as a type.
// If the isDirect attribute of the action is true, then place true on the result output pin
if the input object has the specified classifier as a type.
// Otherwise place false on the result output pin.

ReadIsClassifiedObjectAction action = (ReadIsClassifiedObjectAction)(this.node);

Value input = this.takeTokens(action.object).getValue(0);

boolean result = false;
if (action.isDirect) {
     result = input.hasType(action.classifier);
} else {
     result = input.isInstanceOf(action.classifier);
}
```

8.10.2.30 ReadLinkActionActivation

A read link action activation is a link action activation for a read link action.

Generalizations

- LinkActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the extent, at the current execution locus, of the association to which the action applies.
// For all links that match the link end data, place the value of the remaining "open" end on the result pin.

ReadLinkAction action = (ReadLinkAction)(this.node);
LinkEndDataList endDataList = action.endData;
LinkEndData openEnd = null;

int i = 1;
while((openEnd == null) & i <= endDataList.size()) {
     if (endDataList.getValue(i-1).value == null) {
          openEnd = endDataList.getValue(i-1);
     }
     i = i + 1;
}

ExtensionalValueList extent = this.getExecutionLocus().getExtent(this.getAssociation());

FeatureValueList featureValues = new FeatureValueList();
for (int j = 0; j < extent.size(); j++) {
     ExtensionalValue value = extent.getValue(j);
     Link link = (Link)value;
     if (this.linkMatchesEndData(link, endDataList)) {
          FeatureValue featureValue = link.getFeatureValue(openEnd.end);
          if (!openEnd.end.multiplicityElement.isOrdered | featureValues.size() == 0) {
              featureValues.addValue(featureValue);
          } else {
              int n = featureValue.position;
              boolean continueSearching = true;
              int k = 0;
              while (continueSearching & k < featureValues.size()) {
                  k = k + 1;
                  continueSearching = featureValues.getValue(k-1).position < n;
              }
              if (continueSearching) {
                    featureValues.addValue(featureValue);
                } else {
                    featureValues.addValue(k-1, featureValue);
                }
            }
        }
    }

    for (int j = 0; j < featureValues.size(); j++) {
        FeatureValue featureValue = featureValues.getValue(j);
        this.putToken(action.result, featureValue.values.getValue(0));
    }

    // Now that matching is done, ensure that all tokens on end data input pins
    // are consumed.
    for (int k=0; k<endDataList.size(); k++) {
        LinkEndData endData = endDataList.getValue(k);
        if (endData.value != null) {
            this.takeTokens(endData.value);
        }
    }
```

8.10.2.31 ReadSelfActionActivation

A read self action activation is an action activation for a read self action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the context object of the activity execution containing this action activation and
place a reference to it on the result output pin.

Reference context = new Reference();
```

```
context.referent = this.getExecutionContext();

OutputPin resultPin = ((ReadSelfAction)(this.node)).result;
this.putToken(resultPin, context);
```

8.10.2.32 ReadStructuralFeatureActionActivation

A read structural feature action activation is an action activation for a read structural feature action.

Generalizations

- StructuralFeatureActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value of the object input pin.
// If the given feature is an association end, then get all values of
// that end for which the opposite end has the object input value and
// place them on the result pin.
// Otherwise, if the object input value is a structural value, then get
// the values of the appropriate feature of the input value and place
// them on the result output pin.

ReadStructuralFeatureAction action = (ReadStructuralFeatureAction)(this.node);
StructuralFeature feature = action.structuralFeature;
Value value = this.takeTokens(action.object).getValue(0);
this.putTokens(action.result, this.getValues(value, feature));
```

8.10.2.33 ReclassifyObjectActionActivation

A reclassify object activation is an action activation for a reclassify object action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] checkForMissingFeature (in features : StructuralFeature [0..*], in feature : StructuralFeature) : Boolean

```
boolean isMissing = true;

int i = 1;
while (isMissing & i <= features.size()) {
    StructuralFeature containedFeature = features.getValue(i-1);
    isMissing = containedFeature != feature;
    i = i + 1;
}

return isMissing;
```

[2] doAction ( )

```
// Get the value of the object input pin. If it is not a reference, then do nothing. Otherwise, do the following.
// Remove all types from the referent object that are in the set of old classifiers but not the set of new classifiers (or just all types that are not new classifiers, if isReplaceAll is true).
// Remove the feature values from the referent object for all classifiers that are removed.
// Add all new classifiers as types of the referent object that are not already types.
// Add feature values to the referent object for the structural
// features of all added classifiers.
// Any features that previously had values maintain those values,
// while new features are initialized as being empty.

ReclassifyObjectAction action = (ReclassifyObjectAction)(this.node);
ClassifierList newClassifiers = action.newClassifier;
ClassifierList oldClassifiers = action.oldClassifier;

Value input = this.takeTokens(action.object).getValue(0);

if (input instanceof Reference) {
    Object_ object = ((Reference)input).referent;

    StructuralFeatureList oldFeatures = object.getStructuralFeatures();
    int i = 1;
    while (i <= object.types.size()) {
        Class_ type = object.types.getValue(i-1);

        boolean toBeRemoved = true;
        int j = 1;
        while (toBeRemoved & j <= newClassifiers.size()) {
            toBeRemoved = (type != newClassifiers.getValue(j-1));
            j = j + 1;
         }

         if (toBeRemoved & !action.isReplaceAll) {
             boolean notInOld = true;
             int k = 1;
             while (notInOld  & k <= oldClassifiers.size()) {
                 notInOld = (type != oldClassifiers.getValue(k-1));
                 k = k + 1;
            }
            toBeRemoved = !notInOld;
        }

        if (toBeRemoved) {
            object.types.removeValue(i-1);
        } else {
            i = i + 1;
        }
    }

    for (int n = 0; n < newClassifiers.size(); n++) {
        Classifier classifier = newClassifiers.getValue(n);

        boolean toBeAdded = true;
        int j = 1;
        while (toBeAdded & j <= object.types.size()) {
            toBeAdded = (classifier != object.types.getValue(j-1));
            j = j + 1;
        }

        if (toBeAdded) {
            object.types.addValue((Class_)classifier);
        }
    }
    FeatureValueList oldFeatureValues = object.getFeatureValues();
    object.featureValues = new FeatureValueList();
    object.addFeatureValues(oldFeatureValues);

     // Destroy links involving association ends that were previously features
     // but no longer have feature values after the reclassification.
     StructuralFeatureList newFeatures = object.getStructuralFeatures();
     for (int j = 0; j < oldFeatures.size(); j++) {
         StructuralFeature feature = oldFeatures.getValue(j);
         Association association = this.getAssociation(feature);
         if (association != null) {
             if (this.checkForMissingFeature(newFeatures, feature)) {
                 LinkList links = this.getMatchingLinks(association, feature, input);
                 for (int k = 0; k < links.size(); k++) {
                     Link link = links.getValue(k);
                     link.destroy();
                 }
             }
         }
     }
}
```

8.10.2.34 ReduceActionActivation

A reduce action activation is an action activation for a reduce action.

Generalizations

- ActionActivation

Attributes

None

Associations

- currentExecution : Execution [0..1]
  The current execution of the reducer behavior.

Operations

[1] doAction ( )

```
// Get the values of the collection input pin.
// If the input pin has no values, then do nothing. Otherwise, do the following.
// Repeatedly invoke the reducer behavior on successive pairs to reduce the collection to a
single value, and place that value on the result pin.
```

```
// To invoke the reducer behavior, compile it to create an execution, make the execution the
current execution, place the appropriate values on its input parameters, and execute it.

ReduceAction action = (ReduceAction)(this.node);

ValueList values = this.takeTokens(action.collection);

if (values.size() > 0) {
    ParameterList parameters = action.reducer.ownedParameter;
    Parameter input1 = null;
    Parameter input2 = null;
    Parameter output = null;

    int i = 1;
    while (i <= parameters.size()) {
        Parameter parameter = parameters.getValue(i-1);
        if (parameter.direction == ParameterDirectionKind.in) {
            if (input1 == null) {
                input1 = parameter;
            }
            else {
                input2 = parameter;
            }
        }
        else if (parameter.direction == ParameterDirectionKind.out |
                parameter.direction == ParameterDirectionKind.return ) {
            output = parameter;
        }
        i = i + 1;
    }

    ParameterValue parameterValue1 = new ParameterValue();
    parameterValue1.parameter = input1;
    parameterValue1.values = new ValueList();
    parameterValue1.values.addValue(values.getValue(0));

    int j = 2;
    while (j <= values.size()) {
        this.currentExecution =
this.getExecutionLocus().factory.createExecution(action.reducer,  this.getExecutionContext());
```

```
            this.currentExecution.setParameterValue(parameterValue1);

            ParameterValue parameterValue2 = new ParameterValue();
            parameterValue2.parameter = input2;
            parameterValue2.values = new ValueList();
            parameterValue2.values.addValue(values.getValue(j-1));
            this.currentExecution.setParameterValue(parameterValue2);

            this.currentExecution.execute();

            parameterValue1.values  =  this.currentExecution.getParameterValue(output).values;

            j = j + 1;

            if (parameterValue1.values.isEmpty() & j <= values.size()) {
                 parameterValue1.values.add(values.getValue(j - 1));
                 j = j + 1;
            }
     }

     this.putTokens(action.result, parameterValue1.values);
}
```

[2] terminate ( )

```
// If there is a current execution, terminate it. Then terminate self.

if (this.currentExecution != null) {
     this.currentExecution.terminate();
}

super.terminate();
```

8.10.2.35 RemoveStructuralFeatureValueActionActivation

A remove structural feature action activation is a write structural feature action activation for a remove structural feature value action.

Generalizations

- WriteStructuralFeatureActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the values of the object and value input pins.
// If the given feature is an association end, then destroy any matching links.
// Otherwise, if the object input is a structural value, remove values from the given
feature.
// If isRemoveDuplicates is true, then destroy all current matching links or remove all
values equal to the input value.
// If isRemoveDuplicates is false and there is no removeAt input pin, remove any one feature
value equal to the input value (if there are any that are equal).
// If isRemoveDuplicates is false, and there is a removeAt input pin remove the feature value
at that position.

RemoveStructuralFeatureValueAction action = (RemoveStructuralFeatureValueAction)(this.node);
StructuralFeature feature = action.structuralFeature;
Association association = this.getAssociation(feature);

Value value = this.takeTokens(action.object).getValue(0);

Value inputValue = null;
if (action.value != null) {
    // NOTE: Multiplicity of the value input pin is required to be 1..1.
    inputValue = this.takeTokens(action.value).getValue(0);
}

int removeAt = 0;
if (action.removeAt != null) {
    removeAt =
((UnlimitedNaturalValue)this.takeTokens(action.removeAt).getValue(0)).value.naturalValue;
}

if (association != null) {
    LinkList links = this.getMatchingLinksForEndValue(association, feature, value,
inputValue);

    if (action.isRemoveDuplicates) {
        for (int i = 0; i < links.size(); i++) {
            Link link = links.getValue(i);
            link.destroy();
        }

    } else if (action.removeAt == null) {
         // *** If there is more than one matching link, non-deterministically choose one. ***
        if (links.size() > 0) {
            int i =
((ChoiceStrategy)this.getExecutionLocus().factory.getStrategy("choice")).choose(links.size())
;
             links.getValue(i-1).destroy();
        }

    } else {
        boolean notFound = true;
        int i = 1;
        while (notFound & i <= links.size()) {
            Link link = links.getValue(i-1);
             if (link.getFeatureValue(feature).position == removeAt) {
                notFound = false;
                link.destroy();
            }
            i = i + 1;
        }
    }

} else if (value instanceof StructuredValue) {

     // If the value is a data value, then it must be copied before
     // any change is made.
     if (!(value instanceof Reference)) {
          value = value.copy();
     }

    FeatureValue featureValue =
((StructuredValue)value).getFeatureValue(action.structuralFeature);

    if (action.isRemoveDuplicates) {
         int j = this.position(inputValue, featureValue.values, 1);
        while (j > 0) {
             featureValue.values.remove(j-1);
             j = this.position(inputValue, featureValue.values, j);
        }
    } else if (action.removeAt == null) {
        intList positions = new intList();
        int j = this.position(inputValue, featureValue.values, 1);
        while (j > 0) {
           positions.addValue(j);
           j = this.position(inputValue, featureValue.values, j + 1);
        }

        if (positions.size()>0) {
            // *** Nondeterministically choose which value to remove. ***
            int k = 
((ChoiceStrategy)this.getExecutionLocus().factory.getStrategy("choice")).choose(positions.siz
e());
            featureValue.values.remove(positions.getValue(k-1) - 1);
        }

    } else {
        if (featureValue.values.size() >= removeAt) {
            featureValue.values.remove(removeAt-1);
        }
    }
}

if (action.result != null) {
    this.putToken(action.result, value);
}
```

8.10.2.36 ReplyActionActivation

A reply action activation is an action activation for a reply action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Reply to the call represented by the return information on
// the return information pin using the reply values given
// on the reply value pins.

ReplyAction action = (ReplyAction) this.node;
Trigger replyToCall = action.replyToCall;
InputPinList replyValuePins = action.replyValue;
InputPin returnInformationPin = action.returnInformation;

ValueList values = this.takeTokens(returnInformationPin);
ReturnInformation returnInformation = (ReturnInformation) values.getValue(0);

if (replyToCall.event instanceof CallEvent &
          ((CallEvent)replyToCall.event).operation ==
          returnInformation.getOperation()) {

     ParameterValueList parameterValues = new ParameterValueList();
     int i = 1;
     while (i <= replyValuePins.size()) {
          ParameterValue parameterValue = new ParameterValue();
          parameterValue.values = this.takeTokens(replyValuePins.getValue(i - 1));
          parameterValues.addValue(parameterValue);
          i = i + 1;
     }

     returnInformation.reply(parameterValues);
}
```

8.10.2.37 ReturnInformation

Return information is a value that contains the information necessary to return from an operation call handled as a call event in an activity. It is placed on the returnInformation output pin of an accept call action and is only usable as a value on the input pin of a reply action.

Generalizations

- Value

Attributes

None

Associations

- callEventOccurrence : CallEventOccurrence
  The call event occurrence for the call to which the return information applies.

Operations

[1] getOperation ( ) : Operation

```
// Return the operation associated with the call event occurrence of this
// return information.

return this.callEventOccurrence.getOperation();
```

[2] copy ( ) : Value

```
// Create a new return information value that is a copy of this value, with
// the same call event occurrence.

ReturnInformation copy = (ReturnInformation)super.copy();
copy.callEventOccurrence = this.callEventOccurrence;
return copy;
```

[3] equals(Value otherValue) : Boolean

```
// One return information value equals another if they are for the
// same call event occurrence.

boolean isEqual = false;

if (otherValue instanceof ReturnInformation) {
     isEqual = ((ReturnInformation)otherValue).callEventOccurrence ==
               this.callEventOccurrence;
}

return isEqual;
```

[4] getTypes ( ) : Classifier\[0..*\]

```
// Return information is untyped.

return new ClassifierList();
```

[5] new_ ( ) : Value

```
// Create a new return information value, with an empty call event occurrence.
```

```
return new ReturnInformation();
```

[6] reply (outputParameterValues : ParameterValue[0..*])

```
// Reply to the call by setting the output parameters and
// releasing the caller.

this.callEventOccurrence.setOutputParameterValues(outputParameterValues);
this.callEventOccurrence.returnFromCall();
```

[7] specify ( ) : ValueSpecification

```
// Return information cannot be specified using a value specification.

return null;
```

[8] toString ( ) : String

```
// Return a string representation of the return information.

String s = "ReturnInformation";
String name = this.callEventOccurrence.getOperation().name;
if (name != null) {
     s = s + "(" + name + ")";
}
return s;
```

8.10.2.38 SendSignalActionActivation

A send signal action activation is an invocation action activation for a send signal action.

Generalizations

- InvocationActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value from the target pin. If the value is not a reference, then do nothing.
// Otherwise, construct a signal using the values from the argument pins and send it to the
referent object.

SendSignalAction action = (SendSignalAction)(this.node);
Value target = this.takeTokens(action.target).getValue(0);

if (target instanceof Reference) {
     Signal signal = action.signal;

     SignalInstance signalInstance = new SignalInstance();
     signalInstance.type = signal;

     PropertyList attributes = signal.ownedAttribute;
     InputPinList argumentPins = action.argument;
     for (int i = 0; i < attributes.size(); i++) {
          Property attribute = attributes.getValue(i);
          InputPin argumentPin = argumentPins.getValue(i);
          ValueList values = this.takeTokens(argumentPin);
          signalInstance.setFeatureValue(attribute, values, 0);
     }

     SignalEventOccurrence signalEventOccurrence = new SignalEventOccurrence();
     signalEventOccurrence.signalInstance = (SignalInstance) signalInstance.copy();
     signalEventOccurrence.sendTo((Reference)target);
}
```

8.10.2.39 StartClassifierBehaviorActionActivation

A start classifier behavior action activation is an action activation for a start classifier behavior action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value on the object input pin. If it is not a reference, then do nothing.
// Start the classifier behavior of the referent object for the classifier given as the type
of the object input pin.
// If the object input pin has no type, then start the classifier behaviors of all types of
the referent object. [The required behavior in this case is not clear from the spec.]

StartClassifierBehaviorAction action = (StartClassifierBehaviorAction)(this.node);

Value object = this.takeTokens(action.object).getValue(0);

if (object instanceof Reference) {
     ((Reference)object).startBehavior((Class_)(action.object.typedElement.type), new
ParameterValueList());
}
```

8.10.2.40 StartObjectBehaviorActionActivation

A start behavior action activation is an action activation for a start behavior action.

Generalizations

- InvocationActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value on the object input pin. If it is not a reference, then do nothing.
// Start the behavior of the referent object for the classifier given as the type of the
object input pin, with parameter values taken from the argument input pins.
// If the object input pin has no type, then start the classifier behaviors of all types of
the referent object.

StartObjectBehaviorAction action = (StartObjectBehaviorAction)(this.node);

Value object = this.takeTokens(action.object).getValue(0);

if (object instanceof Reference) {
    Class_ type = (Class_)(action.object.typedElement.type);
    InputPinList argumentPins = action.argument;
    ParameterValueList inputs = new ParameterValueList();

    if (type != null) {
        Behavior behavior;

        if (type instanceof Behavior) {
            behavior = (Behavior)type;
        } else {
            behavior = type.classifierBehavior;
        }

        if (behavior != null) {
            ParameterList parameters = behavior.ownedParameter;

            int pinNumber = 1;
            int i = 1;
            while (i <= parameters.size()) {
                Parameter parameter = parameters.getValue(i-1);
                int j = pinNumber;
                if (parameter.direction == ParameterDirectionKind.in |
                    parameter.direction == ParameterDirectionKind.inout) {
                    ParameterValue parameterValue = new ParameterValue();
                    parameterValue.parameter = parameter;
                    parameterValue.values = this.takeTokens(argumentPins.getValue(j-1));
                    inputs.addValue(parameterValue);
                    j = j + 1;
                }
                pinNumber = j;
                i = i + 1;
            }
        }
    }

    ((Reference)object).startBehavior(type, inputs);
}
```

8.10.2.41 StructuralFeatureActionActivation

A structural feature action activation is an action activation for a structural feature action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

None

8.10.2.42 StructuredActivityNodeActivation

A structured activity node activation is an action activation for an action that is a structured activity node.

Generalizations

- ActionActivation

Attributes

None

Associations

- activationGroup : ActivityNodeActivationGroup
  The group of activations of the activity nodes contained in the structured activity node.

Operations

[1] completeAction(): Token[*]

```
// Only actually complete this structured activity node if it is not
// suspended.

TokenList incomingTokens = new tokenList();
if (!this.isSuspended()) {
     incomingTokens = super.completeAction();
}
return incomingTokens;
```

[2] createEdgeInstances ( )

```
// Create instances for all edges owned by this node.

this.activationGroup.createEdgeInstances(((StructuredActivityNode)(this.node)).edge);
```

[3] createNodeActivations ( )

```
// Create an activation group and create node activations for all the nodes within the
structured activity node.
```

```
super.createNodeActivations();

this.activationGroup = new ActivityNodeActivationGroup();
this.activationGroup.containingNodeActivation = this;
this.activationGroup.createNodeActivations(((StructuredActivityNode)(this.node)).node);
```

[4] doAction ( )

```
// If the structured activity node has mustIsolate=true, then carry out its behavior with isolation.
// Otherwise just activate it normally.

if (((StructuredActivityNode)(this.node)).mustIsolate) {
     _beginIsolation();
          this.doStructuredActivity();
     _endIsolation();
} else {
     this.doStructuredActivity();
}
```

[5] doStructuredActivity ( )

```
// Run all activations of contained nodes. When this is complete, return.
// (This is the default behavior for a structured activity node used simply as a group. It is overridden for the execution of conditional and loop nodes.)

Action action = (Action)(this.node);

// *** Concurrently send offers from all input pins. ***
InputPinList inputPins = action.input;
for (Iterator i = inputPins.iterator(); i.hasNext();) {
     InputPin inputPin = (InputPin)i.next();
     PinActivation pinActivation = this.getPinActivation(inputPin);
     pinActivation.sendUnofferedTokens();
}

this.activationGroup.run(this.activationGroup.nodeActivations);
```

[6] getNodeActivation (in node : ActivityNode) : ActivityNodeActivation [0..1]

```
// If this structured activity node activation is not for the given node, then check if there is an activation for the node in the activation group.
```

```
ActivityNodeActivation thisActivation = super.getNodeActivation(node);

ActivityNodeActivation activation = null;
if (thisActivation != null) {
     activation = thisActivation;
} else if (this.activationGroup != null) {
     activation = this.activationGroup.getNodeActivation(node);
}

return activation;
```

[7] getPinValues (in pin : OutputPin) : Value [0..*]

```
// Return the values of the tokens on the pin activation corresponding to the given pin in
the internal activation group for this node activation.

PinActivation pinActivation = (PinActivation)(this.activationGroup.getNodeActivation(pin));
TokenList tokens = pinActivation.getTokens();

ValueList values = new ValueList();
for (int i = 0; i < tokens.size(); i++) {
     Token token = tokens.getValue(i);
     Value value = ((ObjectToken)token).value;
     if (value != null) {
          values.addValue(value);
     }
}

return values;
```

[8] isSourceFor(edgeInstance: ActivityEdgeInstance): Boolean

```
// Returns true if this node is either the source for the given
// edgeInstance itself or if it contains the source in its
// activation group.

boolean isSource = super.isSourceFor(edgeInstance);
if (!isSource) {
     isSource = this.activationGroup.hasSourceFor(edgeInstance);
}
return isSource;
```

[9] isSuspended(): Boolean

```
// Check if the activation group for this node is suspended.

return  this.activationGroup.isSuspended();
```

[10] makeActivityNodeList (in nodes : ExecutableNode [0..*]) : ActivityNode [0..*]

```
// Return an activity node list containing the given list of executable nodes
// and any pins that they own.

ActivityNodeList activityNodes = new ActivityNodeList();

for (int i = 0; i < nodes.size(); i++) {
     ActivityNode node = nodes.getValue(i);
     activityNodes.addValue(node);

     if (node instanceof Action) {
          Action action = (Action)node;

          InputPinList inputPins = action.input;
          for (int j = 0; j < inputPins.size(); j++) {
               InputPin inputPin = inputPins.getValue(j);
               activityNodes.addValue(inputPin);
          }

          OutputPinList outputPins = action.output;
          for (int j = 0; j < outputPins.size(); j++) {
               OutputPin outputPin = outputPins.getValue(j);
               activityNodes.addValue(outputPin);
          }
     }
}

return  activityNodes;
```

[11] putPinValues (in pin : OutputPin, in values : Value [0..*])

```
// Place tokens for the given values on the pin activation corresponding to the given output
pin on the internal activation group for this node activation.

PinActivation  pinActivation  =  (PinActivation)(this.activationGroup.getNodeActivation(pin));
```

```
for (int i = 0; i < values.size(); i++) {
     Value value = values.getValue(i);
     ObjectToken token = new ObjectToken();
     token.value = value;
     pinActivation.addToken(token);
}
```

[12] resume

```
// When this structured activity node is resumed after being suspended,
// then complete its prior firing and, if there are more incoming
// tokens, fire it again. If, after that, the node is not suspended,
// then finish its resumption.

TokenList incomingTokens = super.completeAction();
if (incomingTokens.size() > 0) {
     this.fire(incomingTokens);
}
if (!this.isSuspended()) {
     super.resume();
}
```

[13] terminate ( )

```
// Terminate the execution of all contained node activations (which
// completes the performance of the structured activity node
// activation), and then terminate this node itself.

this.terminateAll();
super.terminate();
```

[14] terminateAll ( )

```
// Terminate the execution of all contained node activations (which
// completes the performance of the structured activity node
// activation).

this.activationGroup.terminateAll();
```

8.10.2.43 TestIdentityActionActivation

A test identity action activation is an action activation for a test identity action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the values from the first and second input pins and test if they are equal. (Note the
equality of references is defined to be that they have identical referents.)
// If they are equal, place true on the pin execution for the result output pin, otherwise
place false.

TestIdentityAction action = (TestIdentityAction)(this.node);

Value firstValue = this.takeTokens(action.first).getValue(0);
Value secondValue = this.takeTokens(action.second).getValue(0);

Value testResult = this.makeBooleanValue(firstValue.equals(secondValue));
this.putToken(action.result, testResult);
```

8.10.2.44 TokenSet

A set of tokens taken from an input pin activation or input expansion node activation for an expansion region.

Generalizations

None

Attributes

None

Associations

- tokens : Token [0..*]
  The set of tokens in this token set.

Operations

None

8.10.2.45 UnmarshallActionActivation

An unmarshall action activation is an action activation for an unmarshall action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Get the value from the object input pin. If it is a structured value,
// get the values for each of its attributes and place them on the
// corresponding result pin. (Note that the number of result pins is
// presumed to be the same as the number of attributes.)

UnmarshallAction action = (UnmarshallAction) this.node;
Classifier unmarshallType = action.unmarshallType;
OutputPinList resultPins = action.result;

Value value = this.takeTokens(action.object).getValue(0);

if (value instanceof StructuredValue) {
     StructuralFeatureList features =
               ((StructuredValue)value).getMemberFeatures(unmarshallType);
     for (int i=0; i < features.size(); i++) {
          StructuralFeature feature = features.getValue(i);
          OutputPin resultPin = resultPins.getValue(i);
          this.putTokens(resultPin, this.getValues(value, feature));
     }}
```

8.10.2.46 Values

Generalizations

None

Attributes

None

Associations

- values : Value [0..*]

Operations

None

8.10.2.47 ValueSpecificationActionActivation

A value specification action activation is an action activation for a value specification action.

Generalizations

- ActionActivation

Attributes

None

Associations

None

Operations

[1] doAction ( )

```
// Evaluate the value specification for the action and place the result on the result pin of
the action.

ValueSpecificationAction action = (ValueSpecificationAction)(this.node);

Value value = this.getExecutionLocus().executor.evaluate(action.value);
this.putToken(action.result, value);
```

8.10.2.48 WriteLinkActionActivation

A write link action activation is a link action activation for a write link action.

Generalizations

- LinkActionActivation

Attributes

None

Associations

None

Operations

None

8.10.2.49 WriteStructuralFeatureActionActivation

A write structural feature action activation is a structural feature action activation for a write structural feature action.

Generalizations

- StructuralFeatureActionActivation

Attributes

None

Associations

None

Operations

[1] position (in value : Value, in list : Value [0..*], in startAt : Integer) : Integer

```
// Return the position (counting from 1) of the first occurrence of the given value in the
given list at or after the starting index, or 0 if it is not found.

boolean found = false;
int i = startAt;
while (!found & i <= list.size()) {
     found = list.getValue(i-1).equals(value);
    i = i + 1;
}

if (!found) {
    i = 1;
}

return i - 1;
```

9 Foundational Model Library

9.1 General

This clause defines the basic Foundational Model Library for fUML. This is a library of user-level model elements that can be referenced in a fUML model. These capabilities are provided in an overall package called FoundationalModelLibrary, with the sub-packages shown in Figure 9.1 and described in the following subclauses.

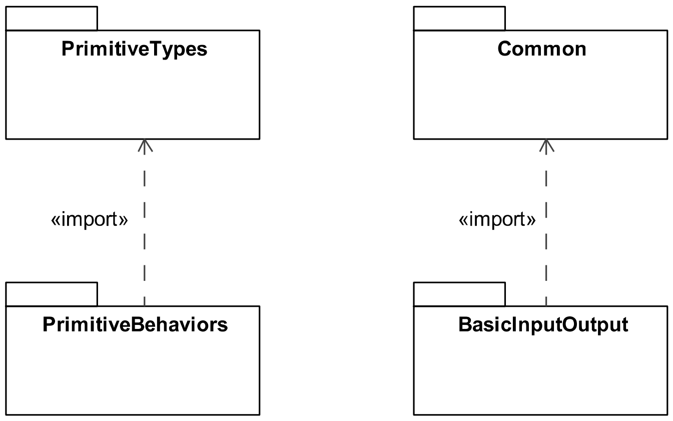


Figure 9.1 - Foundation Model Library packages

9.2 Primitive Types

Note in Figure 9.1 that the PrimitiveBehaviors package imports the PrimitiveTypes package from UML 2 (see Clause 21 of the UML Specification). This package defines the primitive types Boolean, Integer, Real, String, and UnlimitedNatural. These are the types for which corresponding literal values can be specified in fUML (see 7.4). Since they are used in the construction of literal values, these types must all be registered with the execution factory at every locus (see 8.3.1). By importing the PrimitiveTypes package from UML, a user model may also directly reference these types.

Table 9.1 describes the value domains of these primitive types as they are provided in fUML. In the fUML execution model, values of these primitive types are represented by the various subclasses of PrimitiveValue (see 8.6). Each of these subclasses defines a value attribute, which is, itself, typed by a similarly named primitive type (e.g., the type of BooleanValue::value is Boolean). However, the semantics for the primitive types used within the execution model are given by the base semantics in Clause 10. The formalizations of these types in the base semantics provides the grounding for the semantics of the corresponding primitive types as used in fUML models.

Table 9.1 - Primitive Types

Type Name	Description
Boolean	The Boolean type has two literal values, true and false. Note, however, that Boolean is defined as a primitive type, not an enumeration.
Integer	The Integer type has literal values in the (infinite) set of integers (…-2, -1, 0, 1, 2…). However, a conforming implementation may limit the supported values to a finite set.
Real	The Real type has literal values in the infinite, continuous set of real numbers. However, a conforming implementation may limit the support values to a finite subset.
String	The String type has literal values that are sequences of zero or more characters. The actual character set used is not specified in this standard and the maximum string size is unbounded.
UnlimitedNatural	The UnlimitedNatural type has literal values in the (infinite) set of integers (0, 1, 2…) plus the additional value "unbounded." If a conforming implementation limits the set of integers supported, then the set of unlimited natural values supported (other than "unbounded") must be exactly the same as the supported set of non-negative integer values.

9.3 Primitive Behaviors

The FoundationalModelLibrary::PrimitiveBehaviors package contains a set of primitive behaviors that operate on the primitive data types defined in 9.2. As shown in Figure 9.2, the package is divided into sub-packages for each primitive type.

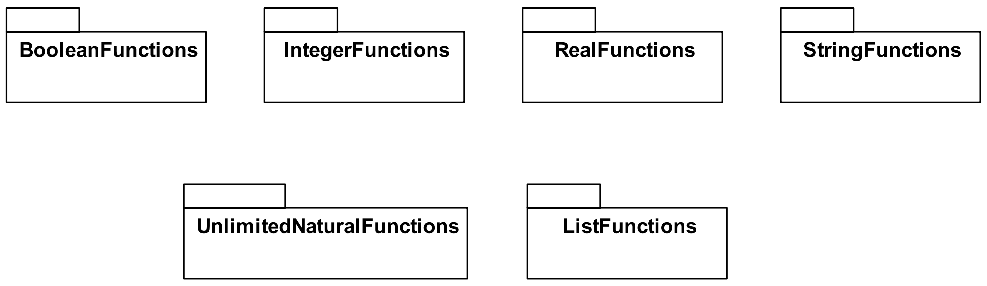


Figure 9.2 - Foundation Model Library PrimitiveBehaviors package

Within each of the sub-packages shown in Figure 9.2, the primitive behaviors are modeled as function behaviors with no side effects. If implemented in the execution environment of a conforming execution tool, implementations for these behaviors are considered to be registered at the locus of execution that models that environment (see 8.3.1). They may be called from user models using the call behavior action (see 7.11).

The primitive behaviors provided in the Foundation Model Library for Boolean, Integer, Real, and String have been largely based on the operations provided for the corresponding primitive types in OCL (see subclauses 11.4 and 11.5 of the OCL Specification). However, while OCL uses an object-oriented operational style for primitive functions (e.g., in binary arithmetic operations, one of the arguments acts as the "target" of the operation invocation), the corresponding behaviors in the Foundation Model Library are invoked more traditionally as functions of all their arguments (particularly since fUML does not allow operations on data types – see 7.6). Other substantive differences from the OCL operations are noted below in the descriptions of the Foundation Model Library behaviors.

Note: An equality function is not provided as a primitive behavior, since this functionality is provided by the test identity action, which tests by value for primitive data types (see 8.10). In particular, since strings are primitive values, the equality test on strings is by value.

In the following descriptions, if the behavior of a primitive behavior can be described in terms of other primitive behaviors, then this is formalized by giving a post-condition for the first behavior. The semantics of primitive behaviors for which no post-condition is given are to be considered to be specified directly by the axioms of the base semantics (see Clause 10).

In some cases, pre-conditions are also specified for primitive behaviors. In this case, if the pre-condition is violated, then the behavior completes execution, but produces no output value. The result parameters for such behaviors are specified to have multiplicity 0..1 to allow for this.

Note: For readability of the pre- and post-condition expressions in the following, an infix notation is used to denote the invocation of binary function behaviors. For example the invocation of the "And" behavior is written "x And y," not "And(x,y)." However, this is still intended to denote the result of the invocation of the named Foundational Model Library primitive behavior on the given arguments. An infix notation "x = y" is also used for equality, with the intended semantics being those of a test identity action on data value arguments (see 8.10).

9.3.1 Boolean Functions

Table 9.2 lists the function behaviors that are included in the package BooleanFunctions. The naming is consistent with OCL, except that names are capitalized, per the usual convention for behaviors (as kinds of classes). The Foundation Model Library also provides ToString and ToBoolean functions not found in OCL.

Table 9.2 - Foundation Model Library Boolean Functions

Function Signature	**Description**
Or(x: Boolean, y: Boolean): Boolean	True if either *x* or *y* is true. *Post: if x then result = true else result = y endif*
Xor(x: Boolean, y: Boolean): Boolean	True if either *x* or *y* is true, but not both. *Post: result = (x Or y) And Not(x And y)*
And(x: Boolean, y: Boolean):Boolean	True if both *x* and *y* are true. *Post: if x then result = y else result = true endif*
Not(x: Boolean): Boolean	True is *x* is false. *Post: if x then result = false else result = true endif*
Implies(x: Boolean, y: Boolean): Boolean	True if *x* is false, or if *x* is true and *y* is true. *Post: result = Not(x) Or (x And y)*
ToString(x: Boolean): String	Converts *x* to a String value. *Post: if x then result = "true" else result = "false" endif*

Table 9.2 - Foundation Model Library Boolean Functions

Function Signature	Description
ToBoolean(x: String): Boolean[0..1]	Converts *x* to a Boolean value. *Pre: (lower(x) = "true") or (lower(x) = "false")* *Post: if lower(x) = "true" then result = true else result = false endif* **Note:** The notation *"lower(x)"* above is *not* intended to be an invocation of a Foundation Model Library primitive behavior but, rather, is intended to denote that value of the string *x* with any uppercase letters converted to the corresponding lowercase letters.

9.3.2 Integer Functions

Table 9.3 lists the function behaviors that are included in the package IntegerFunctions. The naming is consistent with OCL, including the use of the conventional symbols for arithmetic functions, except that the negation function is named "Neg," rather than overloading the symbol "-", and alphabetic names are capitalized, per the usual convention for behaviors (as kinds of classes). The Foundation Model Library also provides ToString and ToUnlimitedNatural functions not found in OCL. The ToInteger function does correspond to an OCL operation, though, in OCL, it is a String operation.

Table 9.3 - Foundational Model Library Integer Functions

Function Signature	Description
Neg(x: Integer): Integer	The negative value of *x.*
+(x: Integer, y: Integer): Integer	The value of the addition of *x* and *y.*
-(x: Integer, y: Integer): Integer	The value of the subtraction of *x* and *y.* *Post: result + y = x*

*(x:Integer, y:Integer): Integer	The value of the multiplication of *x* and *y.*
	
	Post:
	*if y < 0 then result =Neg (x * Neg(y))*
	else if y = 0 then result = 0
	*else result = (x * (y-1)) + x*
	endif endif

/(x: Integer, y: Integer): Real[0..1]	The value of the division of x by y. *Pre: y<>0* *Post: result = ToReal(x) / ToReal(y)* **Note:** The ToReal and "/" functions used here are those from the RealFunctions package (see 9.3.3)
Abs(x: Integer): Integer	The absolute value of x. *Post: if x < 0 then result = Neg(x) else result = x endif*

Table 9.3 - Foundational Model Library Integer Functions

Function Signature	**Description**
Div(x: Integer, y: Integer): Integer[0..1]	The number of times that *y* fits completely within *x.* *Pre: y<>0* *Post:* *if (x * y) >= 0 then* *((result * Abs(y)) <= Abs(x)) And (((result +1) * Abs(y)) > Abs(x))* *else* *((Neg(result) * Abs(y)) <= Abs(x)) And (((Neg(result)+1) * Abs(y)) > Abs(x))* *endif*
Mod(x: Integer, y: Integer): Integer	The result is *x* modulo *y.* *Post: result = x – (x Div y) * y*
Max(x: Integer, y: Integer): Integer	The maximum of *x* and *y.* *Post: if x >= y then result = x else result = y endif*
Min(x: Integer, y: Integer): Integer	The minimum of *x* and *y.* *Post: if x <= y then result = x else result = y endif*
<(x: Integer, y: Integer): Boolean	True if x is less than y.
>(x: Integer, y: Integer): Boolean	True if x is greater than y. *Post: result = Not(x <= y)*
<=(Integer, Integer): Boolean	True if *x* is less than or equal to *y*. *Post: result = (x = y) Or (x < y)*
>=(Integer, Integer): Boolean	True if x is greater than or equal to *y.* *Post: result = (x = y) Or (x > y)*
ToString(x: Integer): String	Converts *x* to a String value. *Post: ToInteger(result) = x*
ToUnlimitedNatural(x: Integer): UnlimitedNatural[0..1]	Converts *x* to an UnlimitedNatural value. *Pre: x >= 0* *Post: ToInteger(result) = x*
ToInteger(x: String): Integer[0..1]	Converts *x* to an Integer value. *Pre: x has the form of a legal integer value*

9.3.3 Real Functions

Table 9.4 lists the function behaviors that are included in the package RealFunctions. The naming is consistent with OCL, including the use of the conventional symbols for arithmetic functions, except that the negation function is named “Neg,” rather than overloading the symbol "-", and alphabetic names are capitalized, per the usual convention for behaviors (as kinds of classes). The Foundation Model Library also provides ToString and ToInteger functions not found in OCL. The ToReal function does correspond to an OCL operation, though, in OCL, it is a String operation.

Table 9.4 - Foundational Model Library Real Functions

Function Signature	**Description**
Neg(x: Real): Real	The negative value of *x*.
+(x: Real,y: Real): Real	The value of the addition of *x* and *y*.
-(x: Real, y: Real): Real	The value of the subtraction of *x* and *y*. *Post: result + y = x*
Inv(x: Real): Real	The inverse (reciprocal) of *x*.
*(x:Real, y:Real): real	The value of the multiplication of *x* and *y*.
/(x: Integer, y: Integer): Real[0..1]	The value of the division of x by y. *Pre: y<>0* *Post: result * y = x*
Abs(x: Real): Real	The absolute value of x. *Post: if x < 0 then result = Neg(x) else result = x endif*
Floor(x: Real): Integer[0..1]	The largest integer that is less than or equal to *x*. *Post: result <= x and result + 1 > x*
Round(x: Real): Integer[0..1]	The integer that is closest to *x*. When there are two such integers, the largest one. *Post: (Abs(x - result) < 0.5 Or ((Abs(x-result) = 0.5 And result > x*
Max(x: Real, y: Real): Real	The maximum of *x* and *y*. *Post: (Abs(x - result) < 0.5 Or ((Abs(x - result) = 0.5 And result > x)*
Min(x: Real, y: Real): Real	The minimum of *x* and *y*. *Post: if x <= y then result = x else result = y endif*
<(x: Real, y: Real): Boolean	True if x is less than y.
>(x: Real, y: Real): Boolean	True if x is greater than y. *Post: result = Not(x <= y)*
<=(Real, Real): Boolean	True if *x* is less than or equal to *y*. *Post: result = (x = y) Or (x < y)*
>=(Real, Real): Boolean	True if *x* is greater than or equal to *y*. *Post: result = (x = y) Or (x > y)*

Table 9.4 - Foundational Model Library Real Functions

Function Signature	Description
ToString(x: Real): String	Converts *x* to a String value. *Post: ToRealresult) = x*
ToInteger(x: Real): Integer	Converts *x* to an Integer value. *Post: if x >= 0 then Floor(x) else Neg(Floor(Neg(x)) endif*
ToReal(x: String): Real[0..1]	Converts *x* to a Real value. *Pre: x has the form of a legal Real value*

The set of Real numbers includes values that cannot be represented in finite precision (e.g., irrational numbers and those rational numbers with infinite repeating digit representations in the base being used). Therefore, implementations are given the following permissions for representing Real numbers and performing computations on them, while still conforming to this specification.

Note: The permissions below are intended, in particular, to allow the conformance of implementations using finite-precision floating-point representations for Real numbers (such as those based on the popular IEEE 754 standard), which still allows for other implementations that may not need to take advantage of all the allowed permissions.

1. A conforming implementation may support only a *limited range* of Real values, such that the absolute value of any supported value is less than or equal to a specified maximum value. If the implementation limits the range of values supported for Integer, then the maximum value specified for Real must be no less than the greatest absolute value of any supported Integer value.
2. A conforming implementation may support only a *restricted value set* for Real, defined as a non-dense subset of the infinite set of rational numbers (such that any bounded interval of this value set contains only a finite set of values) including zero and with no upper or lower bound. If the implementation limits the range of values supported for Integer, then the smallest positive value in the restricted value set shall be at least as small as the reciprocal of the largest supported Integer value.
3. A conforming implementation may provide distinct representations for Real positive zero and Real negative zero. These values shall be considered equal for the purposes of all comparison functions. However, they may be distinguished in certain arithmetic computations (see below).
4. A conforming implementation may include additional special values that are instances of the Real type but are not numeric values (such as infinite values and “not a number” values). Note that, even if included in an implementation of the Real type, none of these special values have any standard literal representation in UML.

The functions in the RealFunctions package are specified in Table 9.4 in terms of the semantics for mathematical Real numbers, as defined in 10.3.1. However, an implementation that takes advantage of some or all of the above permissions may not be able to produce exact results for some computations using these functions. Therefore, conformance to the function behaviors given in Table 9.4 shall be interpreted as follows:

- Since a restricted value set is non-dense and unbounded, any exact value that is not in such a set will be between two values that are in the set. If a conforming implementation supports only a restricted value set, and the result of a computation is not a member of this set, then the implementation may implement the computation as resulting in one of the two values in the restricted value set that the exact result is between. (If the exact value of the computation is non-zero, but the chosen value in the restricted value set is zero, then the computation is said to *underflow*.)
- If a conforming implementation supports only a limited range of values, then a computation that results in an exact value

that is outside that range is said to *overflow*. The implementation may implement an overflowing computation as resulting in a special value (e.g., positive or negative infinity). If not, an overflowing computation shall be implemented as having an empty result.

- If a numeric result is not defined for a call to a primitive function resulting in a Real value, because one of the arguments is a special value or because a precondition is violated, then a conforming implementation may produce a special value for its result. Otherwise, the computation shall be implemented as having an empty result.
- If a conforming implementation supports signed zero, then multiplication of a numeric value by positive zero shall result in positive zero, while multiplication of a numeric value by negative zero shall result in negative zero. A conforming implementation shall not otherwise distinguish between positive and negative zero in any call to a primitive function that is specified as resulting in a numeric result. However, it may distinguish between them if the result is implemented as a special value (e.g., division by negative zero may result in negative infinity).
- Other than as given above, this specification does not define the result of calling a primitive function in which one or more of the arguments is a special value.

9.3.4 String Functions

Table 9.5 function behaviors are included in the package StringFunctions. The naming is consistent with OCL, except that names are capitalized, per the usual convention for behaviors (as kinds of classes). In the Foundation Model Library, ToInteger is provided as an integer function rather than a string operation, and ToReal is not provided because the Foundation Model Library does not support a Real primitive type.

Table 9.5 - Foundational Model Library String Functions

Function Signature	**Description**
Concat(x: String, y: String):String	The concatenation of *x* and *y*. *Post:* *(Size(result) = Size(x) + Size(y)) And* *(Substring(result, 1, Size(x)) = x) And* *(Substring(result, Size(x)+1, Size(result)) = y)*
Size(x: String):Integer	The number of characters in *x*.
Substring(x: String, lower: Integer, upper: Integer): String[0..1]	The substring of *x* starting at character number *lower*, up to and including character number *upper*. Character numbers run from 1 to *Size(x)*. *Pre:* *(1 <= lower) And* *(lower <= upper) And* *(upper <= Size(x))*

9.3.5 UnlimitedNatural Functions

Table 9.6 lists the function behaviors that are included in the package UnlimitedNaturalFunctions. Only comparison and conversion functions are provided. Arithmetic can be performed on UnlimitedNatural values by converting them to Integers. (Arithmetic on the “unbounded” value is thus not defined.)

Table 9.6 - Foundational Model Library UnlimitedNatural Functions

Function Signature	**Description**
Max(x: UnlimitedNatural, y: UnlimitedNatural): UnlimitedNatural	The maximum of *x* and *y.* *Post: if x >= y then result = x else result = y endif*
Min(x: UnlimitedNatural, y: UnlimitedNatural): UnlimitedNatural	The minimum of *x* and *y.* *Post: if x <= y then result = x else result = y endif*
<(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	True if x is less than y. Every value other than "unbounded" is less than "unbounded".
>(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	True if x is greater than y. *Post: result = Not(x <= y)*
<=(UnlimitedNatural, UnlimitedNatural): Boolean	True if *x* is less than or equal to *y*. *Post: result = (x = y) Or (x < y)*
>=(UnlimitedNatural, UnlimitedNatural): Boolean	True if x is greater than or equal to *y.* *Post: result = (x = y) Or (x > y)*
ToString(x: UnlimitedNatural): String	Converts *x* to a String value. The value "unbounded" is represented by the string "*". *Post: ToUnlimitedNatural(result) = x*
ToInteger(x: UnlimitedNatural): Integer[0..1]	Converts *x* to an Integer value. *Pre: x <> unbounded*
ToUnlimitedNatural(x: String): Integer[0..1]	Converts *x* to an Integer value. *Pre: (x has the form of a legal integer value) Or (x = "*")* *Post:* *if x = "*" then result = unbounded* *else result = ToUnlimitedNatural(ToInteger(x))*

9.3.6 List Functions

Table 9.7lists the function behaviors that are included in the ListFunctions package. These are convenience functions for querying values with multiplicity [*]. Note that the *list* arguments for all the list functions are untyped and that the results of ListGet and ListConcat are also untyped.

Note: The functionality of the list functions could actually be implemented as activities. However, it is generally much more convenient to be able to invoke this simple functionality as if it was primitive behavior, rather than having to model it explicitly.

The list functions are also used in the Java to UML Activity Model mapping (see Annex A).

Table 9.7 - Foundational Model Library List Functions

Function Signature	Description
ListSize(list[*] {nonunique}): Integer	Returns cardinality of the input values in the *list*.
ListGet(list[*]{ordered, nonunique}, index: Integer) [0..1]	Returns the value at the position given by *index* in the ordered *list*. Positions run from 1 to *ListSize(list)*. *Pre: (index > 0) And (index <= ListSize(list))*
ListConcat(list1[*] {ordered, nonunique}, list2[*] {ordered, nonunique}) [*] {ordered, nonunique}	Returns the list with all the values of *list1* followed by all the values of *list2*.

9.4 Common

9.4.1 Overview

The FoundationModelLibrary::Common package contains classifiers shown in Figure 9.3. These classifiers are currently only used in the basic input/output model (see 9.5). However, they are considered potentially usable in a wider context in the future, so they have been separated into their own sub-package. They are further described below in 9.4.2.

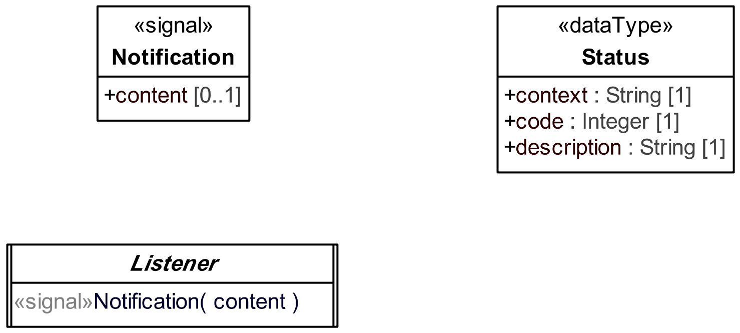


Figure 9.3 - Foundational Model Library Common Package

9.4.2 Classifier Descriptions

9.4.2.1 Listener (active class)

A listener is an active class that can asynchronously receive a notification.

Generalization

None

Receptions

- Notification(content[0..1])

The Listener class declares its ability to receive a Notification signal. Any concrete subclass of Listener should have a classifier behavior that can accept such a signal.

9.4.2.2 Notification (signal)

A notification is a signal used to asynchronously send content to a listener.

Generalization

None

Attributes

- content [0..1]
  An optional value (of any type) sent as the content of the notification.

9.4.2.3 Status (data type)

The Status data type provides a common structure for reporting the normal or error status of a service such as a channel. Operations whose execution may cause an error condition have an optional error status output parameter to report this condition (exceptions are not included in the fUML subset). This output is generated only if there is an error condition-if the operation completes normally, no value is produced. A service may also have an operation to report its current status as of the execution of the last operation on it.

Generalization

None

Attributes

- context: String
  A name (generally a class name) indicating the context in which the status is defined.

- code: Integer
  A numeric status code. A value of zero is the default for normal operation. A value less than zero indicates an error condition. A value greater than zero indicates an informational status condition. Status codes must be unique within a given context, but not necessarily across contexts.

- description: String
  A textual description of the status condition.

9.5 Basic Input/Output

This subclause defines basic capabilities for input and output, provided by set of classes that can be directly referenced from a user model. While this is thus a library of user-model classes, not classes within the execution model itself, the methods implementing the operations of these classes must be provided as primitive capabilities as part of any actual implementation of the library model.

The primary goal of the basic library defined here is to provide a simple semantic foundation for what it means to receive input into and send output from an executing model. It is not intended to be a complete input/output library, but, rather, to act as the underpinning for a more sophisticated future library. Nevertheless, in addition to the foundational input/output mechanism, it does also include a basic set of standard capabilities that allow the expected baseline of textual input and output.

9.5.1 The Channel Model

Within the context of a single model, all communication is between known source and target elements within the model. In this sense, an executing model is a "closed universe." Input and output is, in effect, a controlled means for providing "openings" in this universe for communications in which the actual source or target is not known within the model.

The fundamental abstraction providing for making these "openings" is that of the channel. The basic library model of channels is provided in the package FoundationalModelLibrary::BasicInputOutput. Figure 9.4 shows the classifiers included in this package, which are further described in 9.5.3.

An input channel provides a means for receiving values into an executing model from outside of it. Conversely, an output channel is a means for sending values out of an executing model. An active channel is like an input channel, except that it allows clients to receive input values asynchronously, instead of requesting them synchronously. In addition to the fundamental channel classes, specializations are provided for basic textual input and output capabilities.

Note that all the classes in this model are abstract. They are not intended to be directly instantiated by a user model. Instead, channels must be made available as or by "services" available at the current execution locus (see 8.3 for a discussion of loci and system services). For example, there may be at most one instance of the class StandardInputChannel and one instance of the class StandardOutputChannel "pre-instantiated" at each locus. Or a locus may provide a file service that is used to obtain channels that connect to an external file system.

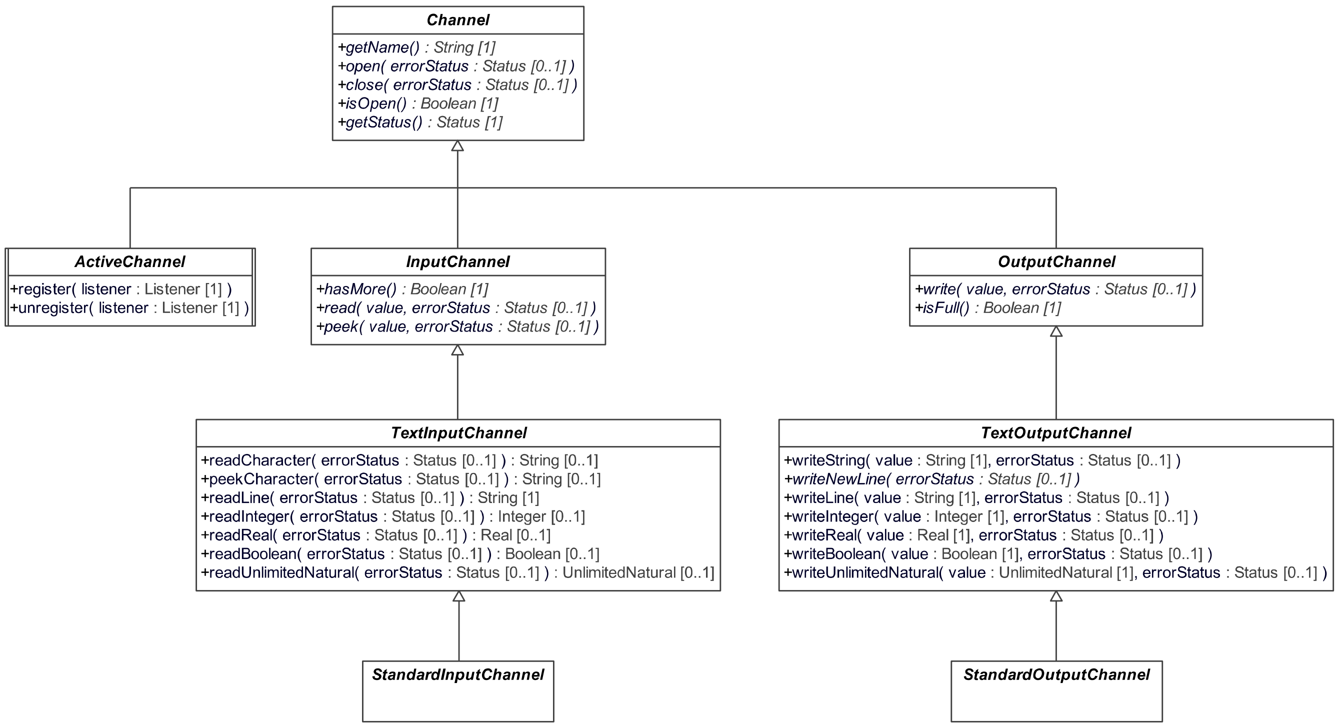

Channel
+getName() : String [1]
+open(errorStatus : Status [0..1])
+close(errorStatus : Status [0..1])
+isOpen() : Boolean [1]
+getStatus() : Status [1]
ActiveChannel
+register(listener : Listener [1])
+unregister(listener : Listener [1])
InputChannel
+hasMore() : Boolean [1]
+read(value, errorStatus : Status [0..1])
+peek(value, errorStatus : Status [0..1])
OutputChannel
+write(value, errorStatus : Status [0..1])
+isFull() : Boolean [1]
TextInputChannel
+readCharacter(errorStatus : Status [0..1]) : String [0..1]
+peekCharacter(errorStatus : Status [0..1]) : String [0..1]
+readLine(errorStatus : Status [0..1]) : String [1]
+readInteger(errorStatus : Status [0..1]) : Integer [0..1]
+readReal(errorStatus : Status [0..1]) : Real [0..1]
+readBoolean(errorStatus : Status [0..1]) : Boolean [0..1]
+readUnlimitedNatural(errorStatus : Status [0..1]) : UnlimitedNatural [0..1]
TextOutputChannel
+writeString(value : String [1], errorStatus : Status [0..1])
+writeNewLine(errorStatus : Status [0..1])
+writeLine(value : String [1], errorStatus : Status [0..1])
+writeInteger(value : Integer [1], errorStatus : Status [0..1])
+writeReal(value : Real [1], errorStatus : Status [0..1])
+writeBoolean(value : Boolean [1], errorStatus : Status [0..1])
+writeUnlimitedNatural(value : UnlimitedNatural [1], errorStatus : Status [0..1])
StandardInputChannel
StandardOutputChannel

Figure 9.4 - Foundational Model Library BasicInputOutput Package: Channel Model

9.5.2 Pre-defined ReadLine and WriteLine Behaviors

The BasicInputOutput package also includes two pre-defined convenience behaviors, ReadLine and WriteLine, which simplify textual input and output to the standard input and output channels. These behaviors may be called using the call behavior action (see 7.11). Both of these behaviors may be formally defined as activities in terms of the functionality provided by the standard channel classes. Figure 9.5 shows an activity definition for ReadLine, and Figure 9.6 shows the definition for WriteLine. However, a conforming implementation may alternatively provide these as primitive behaviors with equivalent functionality.

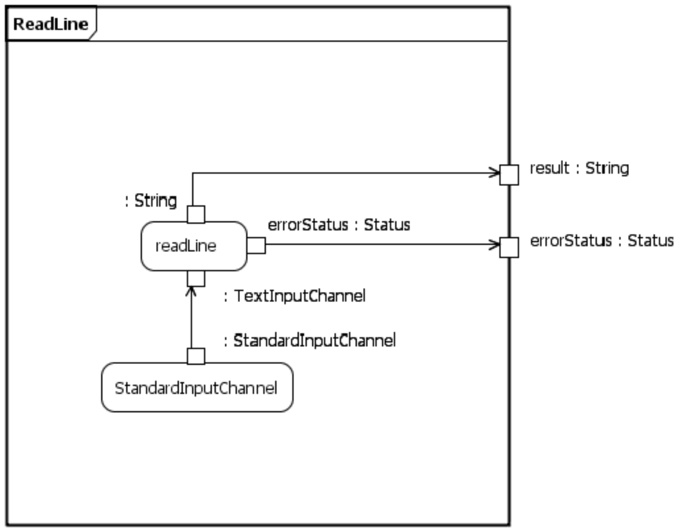


Figure 9.5 - An Activity Definition for the ReadLine Behavior

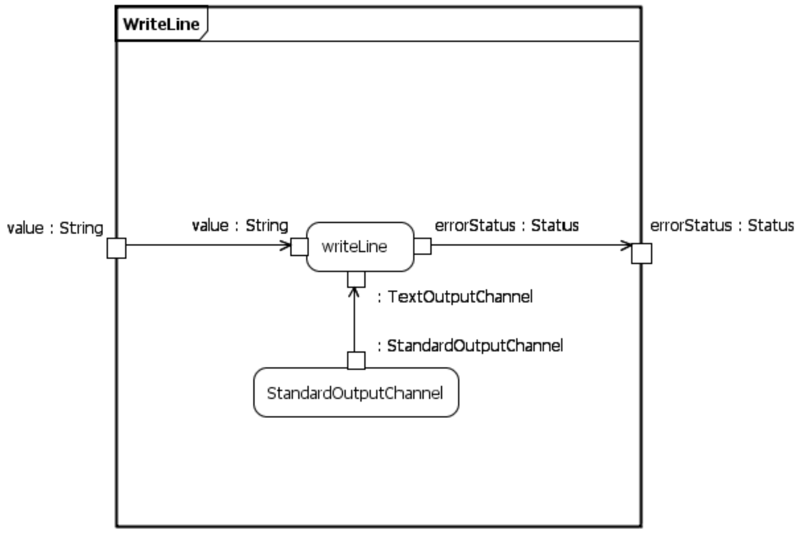


Figure 9.6 - An Activity Definition for the WriteLine Behavior

9.5.3 Class Descriptions

9.5.3.1 ActiveChannel (active class)

An active channel is similar to an input channel, in that it is used to receive input into a model. However, instead of providing input in response to synchronous requests of clients, it allows clients to register as listeners for asynchronous notification of input as it arrives. As each input value arrives, it is sent as the content of a Notification signal to all registered listeners.

Generalization

- Channel

Additional Operations

- register(listener: Listener)
  The *register* operation is used to register a listener with an active channel. If the listener is already registered, then the operation has no effect.

- unregister(listener: Listener)
  The *unregister* operation is used to remove the registration of a listener with an active channel. If the given listener is not registered with the channel, then the operation has no effect.

9.5.3.2 Channel

A channel is a means for receiving or sending values. A channel object within an executing model represents the end of the channel accessible to that model. What is at the other end of the channel—that is the source for input or the target for output —depends on the implementation of a specific channel and is not defined by the library model.

Two standard status codes are defined for every channel, as given in the table below. Additional status codes may be defined for specific kinds of channels (for example, see the descriptions below of the subclasses of Channel). In all cases, the name of the class in which the status is defined below is used as the context for the status code.

Code	Description	Definition
0	Normal	The default for normal operation of the channel.
-1	Not open	The last operation performed on the channel required the channel to be open, but the channel was closed. (All read and write operations of any sort on a channel require the channel to be open.)

Generalization

None

Operations

- getName(): String
  Each channel has a name. How the name of a channel is determined depends on the implementation of a specific channel, but every channel instantiated at a given locus is required to have a different name. The *getName* operation returns the name of the channel.

- open(out errorStatus: Status[0..1])
  A channel may be either *open* or *closed.* Attempting to receive input or send output on a closed channel has no effect. The *open* operation is used to open a channel that is closed. Opening a channel that is already open leaves the channel open and has no other effect.
- close(out errorStatus: Status[0..1])
  The *close* operation closes a channel that is open. Closing a channel that is already closed leaves the channel closed and has no other effect.
- isOpen(): Boolean
  The *isOpen* operation returns true if a channel is open and false if it is closed.
- getStatus(): Status
  The *getStatus* operation returns the current status of a channel (see the description of Status below).

9.5.3.3 InputChannel

An input channel is a channel for receiving input values into a model. The following additional status code is defined for input channels.

Code	Description	Definition
-2	No input	A read operation was attempted, but no more input is currently available on this channel.

Generalization

Channel

Additional Operations

- hasMore(): Boolean
  The *hasMore* operation returns true if there is a value available to be read from an input channel and false if there is not. The operation returns false if the channel is not open. It is an error to attempt to read from an input channel that does not have an available input.
- read(out value[0..1], out errorStatus: Status[0..1])
  The *read* operation is used to obtain an input value from an input channel. The operation has a value out parameter that has no type, which means that it may return a value of any type. If the read operation completes without producing an output value, then the error status is required to have an error value indicating the reason for this.
- **Note:** The read operation uses an out parameter rather than a return result because there is no UML surface syntax for displaying an operation with a return parameter that does not have a type.
- peek(out value[0..1], out errorStatus: Status[0..1])
  The *peek* operation has the same behavior as *read,* except that the value returned is not consumed from the input channel. That is, if the channel has an available value, multiple sequential *peek* calls will continue to return that same value, without removing it from the channel, until the *read* operation is called.

9.5.3.4 OutputChannel

An output channel is a channel for sending output values out of a model. The following additional status codes are defined for output channels.

Code	Description	Definition
-2	Full	A write operation was attempted, but the channel is not able to accept any further output.
-3	Type not supported	A write operation was attempted for a value of a type that is not supported by the channel.

Generalization

Channel

Additional Operations

- isFull(): Boolean
  The *isFull* operation returns false if an output channel is able to accept more output values and true if it is not. The operation returns true if the channel is not open. It is an error to attempt to write to an output channel that is full.
- write(value, out errorStatus: Status[0..1])
  The *write* operation is used to send an output value on an output channel. The operation has a single parameter. This parameter has no type, which means that it may be a value of any type. If the channel is full, then attempting a write operation is an error condition, but the operation is still required to complete its execution (which will have no other effect than to return the appropriate error status).

9.5.3.5 StandardInputChannel

A standard input channel is a text input channel that may be provided as a pre-instantiated service at a locus. Any locus may have at most one instance of the StandardInputChannel class, with the name "StandardInput." Since there can be at most one instance, this instance, if it exists, can be easily obtained by executing a Read Extent action on the StandardInputChannel class.

Generalization

TextInputChannel

Additional Operations

None

9.5.3.6 StandardOutputChannel

A standard output channel is a text output channel that may be provided as a pre-instantiated service at a locus. Any locus may have at most one instance of the StandardOutputChannel class, with the name "StandardOutput." Since there can be at most one instance, this instance, if it exists, can be easily obtained by executing a Read Extent action on the StandardOutputChannel class.

Generalization

TextOutputChannel

Additional Operations

None

9.5.3.7 TextInputChannel

A text input channel is an input channel whose values are text characters. A *read* operation on a text input channel will always return a string value that contains a single character. The additional operations on a text input channel provide convenient capabilities for reading longer strings of characters and, in some cases, treating them as representations of other primitive values. The following additional status code is defined for text input channels.

Code	Description	Definition
-3	Cannot convert	An attempt was made to read an integer, real, Boolean or unlimited natural, but the characters available from the channel do not conform to the required syntax.

Generalization

InputChannel

Additional Operations

- readCharacter(out errorStatus: Status[0..1]): String[0..1]
  The *readCharacter* operation reads the next value from a text input channel and returns it as a string of a single character. No value is returned if none is available from the channel. This is an error condition.

- peekCharacter(out errorStatus: Status[0..1]): String[0..1]
  The *peekCharacter* operation has the same behavior as *readCharacter,* except that the character returned is not consumed from the text input channel. That is, if a character is available on the channel, multiple *peekCharacter* calls will continue to return that character without removing it from the channel, until some read operation is called.

- readLine(out errorStatus: Status[0..1]): String
  The *readLine* operation continues to read characters from the input channel until the end of a line is reached or there are no more characters available from the channel. The characters read are returned, in order, as a string value. The character encoding of a new line is not defined in this specification. Nevertheless, the new line character(s) are required to be consumed by the *readLine* operation, but they are not included in the returned string. Note that if no character is available from the channel when the operation is called, or if the only character(s) read are the new line character(s), then the operation returns the empty string. This is not an error.

- readInteger(out errorStatus: Status[0..1]): Integer[0..1]
  The *readInteger* operation is used to read a textual representation of an integer and return it as an integer value. The textual syntax for an integer is defined to be an optional '+' or '-' character followed by a string of one or more digits '0' through '9.' All characters are read up to (but not including) the first character that does not conform to the required syntax or until no more characters are available. No value is returned if no characters are available from the channel or if the available characters do not begin with a string that conforms to the required syntax, in which case no values are read from the channel. This is an error condition.

- readReal(out errorStatus: Status[0..1]): Real[0..1]
  The *readReal* operation is used to read a textual representation of a real number and return it as a real value. The textual syntax for a real number is defined to have three parts: an integer part with the syntax of an optionally signed integer (see above); a fraction part consisting of a '.' character followed by zero or more digits '0' through '9'; and an exponent part consisting of an 'e' character or an 'E' character followed by an optionally signed integer. A legal real number must have an integer part and/or a fraction part, and, if there is no integer part, the fraction part must have at least one digit. The exponent part is optional. All characters are read up to (but not including) the first character that does not conform to the required syntax, or until no more characters are available. No value is returned if no characters are available from the channel or if the available characters do not begin with a string that conforms to the required syntax, in which case no values are read from the channel. This is an error condition.

- readBoolean(out errorStatus: Status[0..1]): Boolean[0..1]
  The *readBoolean* operation is used to read a textual representation of a Boolean and return it as a Boolean value. The textual syntax for a Boolean is defined to be either the string "true" or the string "false," or any string obtained by capitalizing some or all of the characters of these strings. All characters are read up to (but not including) the first character that does not conform to the required syntax or until no more characters are available. No value is returned if no characters are available from the channel or if the available characters do not begin with a string that conforms to the required syntax, in which case no values are read from the channel. This is an error condition.

- readUnlimitedNatural(out errorStatus: Status[0..1]): UnlimitedNatural[0..1]
  The *readUnlimitedNatural* operation is used to read a textual representation of an unlimited natural number and return it as an integer value. The textual syntax for an unlimited natural is defined to be either the single character '*' or string of one or more digits '0' through '9.' All characters are read up to (but not including) the first character that does not conform to the required syntax or until no more characters are available. No value is returned if no characters are available from the channel or if the available characters do not begin with a string that conforms to the required syntax, in which case no values are read from the channel. This is an error condition.

9.5.3.8 TextOutputChannel

A text output channel is an output channel whose values are text characters. A *write* operation on a text output channel always places characters onto the channel.

- For a string value, each of the characters in the string is sequentially written to the channel.

- Primitive values of types Integer, Real, Boolean, and UnlimitedNatural are written using the syntax given for the *writeInteger*, *writeReal, writeBoolean*, and *writeUnlimitedNatural* operations as described below.

- Enumeration values are written using the names of the corresponding enumeration literal.

- No standard textual representation is defined for other kinds of values, but it is *not* an error to attempt to write them. The actual representation of such values is determined by the specific implementation of the channel.

If during the execution of any write operation on a text output channel the channel becomes full, then the operation returns immediately. This is an error condition, but, if the operation was writing multiple characters, all characters up to the point the channel became full will have been successfully output to the channel.

Generalization

OutputChannel

Additional Operations

- writeString(value: String, out errorStatus: Status[0..1])
  The *writeString* operation sequentially writes each of the characters in the given string value to a text output channel.

- writeNewLine(out errorStatus: Status[0..1])
  The *writeNewLine* operation writes the character(s) encoding a new line to a text output channel. The character encoding of a new line is not defined in this specification, but is determined by the implementation of a specific channel.

- writeLine(value: String, out errorStatus: Status[0..1])
  The *writeLine* operation writes the given string value to a text output channel, followed by a new line.

- writeInteger(value: Integer, out errorStatus: Status[0..1])
  The *writeInteger* operation is used to write a textual representation of an integer. The textual syntax for an integer is

defined to be an optional '-' character (for a negative integer) followed by a string of one or more digits '0' through '9' (note that no '+' is included for a positive integer).

- writeReal(value: Real, out errorStatus: Status[0..1])
  The *writeReal* operation is used to write a textual representation of a real number. The textual syntax for a real number is defined to be an optional '-' character (for a negative number), followed by a string of one or more digits '0' through '9', optionally followed by a '.' character and one or more digits, optionally followed by an 'E' character followed by an optional '-' character (for a negative exponent) followed by one or more digits.

- writeBoolean(value: Boolean, out errorStatus: Status[0..1])
  The *writeBoolean* operation is used to write a textual representation of a Boolean. The textual syntax for a Boolean is defined to be either the string "true" or the string "false."

- writeUnlimitedNatural(value: UnlimitedNatural, out errorStatus: Status[0..1])
  The *writeUnlimitedNatural* operation is used to write a textual representation of an unlimited natural number. The textual syntax for an unlimited natural is defined to be either the single character '*' (for the "unbounded" value) or string of one or more digits '0' through '9.'

This page intentionally left blank

10 Base Semantics

10.1 Design Rationale

This clause gives semantics for the portion of fUML used in the Java to Activity mapping in Annex A (known as "base UML" or bUML, with semantics known as the "base semantics"). Base UML is expressive enough to define the execution model and must be used when specializing the execution model through explicit variation. The base semantics specifies when particular executions conform to a model defined in bUML. It does not generate executions. In particular, the base semantics does not define a virtual machine to execute models directly. The base semantics is expressed in axioms of first order logic. This has the advantage of being completely explicit, rather than using text to explain the behavior of a virtual machine. This enables automatic determination of whether an execution conforms to the execution model. It has the disadvantage of requiring axioms for the semantic interpretation of all syntactic patterns used in the execution model.

This clause assumes familiarity with these background documents:

- Common Logic Interchange Format (CLIF), the language in which the axioms are written[1].
- Process Specification Language (PSL), a foundational axiomatization of processes.[2]

This clause uses an embedded approach to axiomatization, which enables syntax and semantics to be explicitly related through additional axioms, as compared to a translation that depends on a separate translation language.[3] The clause gives axioms for semantics and any additional syntax needed just for the formalization. The semantic axioms identify a particular syntactical pattern used in the execution model and give it a semantic interpretation. The semantic interpretation is grounded in PSL.[4]

10.2 Conventions

Naming conventions for relations used in this clause are:

- buml: Prefixed names are those of metaclasses and metaproperties in bUML. Metaclasses are formalized as unary predicates that are satisfied when applied to an instance of the metaclass. For example, buml:Activity is a predicate satisfied by activities in the execution model. Metaproperties are formalized as binary predicates that are satisfied when applied to two elements linked by the property, with the first being an instance of the owner of the property and the second being the value. For example, buml:activity is a binary predicate satisfied by a node in an activity, and an activity containing the node, in that order.[5]
- psl: Prefixed names are PSL relations.
- form: Prefixed names are relations introduced only for the formalization.

This clause assumes multiple generalization in bUML.

1 ISO 24707, http://standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007(E).zip.

2 ISO 18629-1:2004, see https://www.iso.org/standard/35431.html for the standard and http://www.conradbock.org/#PSL for introductory material.

3 More about this at http://www.ihmc.us/users/phayes/CL/SW2SCL.html and http://www.w3.org/TR/daml+oil-axioms.

4 The axiomitization of numbers and cardinality is adapted from http://philebus.tamu.edu/cmenzel/Papers/AxiomaticSemantics.pdf.

5 This clause assumes the execution model conforms to the abstract syntax of bUML, including constraints.

Basic additions to PSL used in this clause are:

```
(forall (s occ)
   (iff (form:subactivity-occurrence-neq s occ)
        (and (psl:subactivity_occurrence s occ)
             (not (= s occSuper)))))

(forall (f s)
   (iff (form:priorA f s)
        (exists (sRoot)
           (and (psl:root_occ sRoot s)
                (psl:prior f sRoot)))))

(forall (f s)
   (iff (form:holdsA f s)
        (exists (sLeaf)
           (and (psl:leaf_occ sLeaf s)
                (psl:holds f sLeaf)))))

(forall (s1 s2 a)
   (iff (form:min-precedesA s1 s2 a)
        (exists (s1Leaf s2Root)
           (and (psl:leaf_occ s1Leaf s1)
                (psl:root_occ s2Root s2)
                (psl:min_precedes s1Leaf s2Root a)))))
```

10.3 Structure

This subclause covers the structural aspects of the base semantics.

10.3.1 Primitive Types

This subclause covers primitive types in bUML: Boolean, UnlimitedNatural, Integer, and String, as well as other kinds of numbers and sequences introduced for the formalization.

10.3.1.1 Boolean

```
(forall (x)
   (if (buml:Boolean x)
       (or (= x form:true)
           (= x form:false))))

(not (= form:true form:false))

(forall (x y)
   (if (form:not x y)
       (and (buml:Boolean x)
            (buml:Boolean y))))

(forall (x y)
   (if (form:not x y)
       (not (= x y))))
(forall (x y z)
   (if (form:and x y z)
       (and (buml:Boolean x)
            (buml:Boolean y)
            (buml:Boolean z))))

(forall (x y z)
   (if (form:and x y z)
       (and (iff (= z form:false)
                 (or (= x form:false)
                     (= y form:false)))
            (iff (= z form:true)
                 (and (= x form:true)
                      (= y form:true))))))
```

10.3.1.2 Numbers

The less-than relation.

```
(forall (x y)
  (if (buml:less-than x y)
      (and (buml:Real x)
           (buml:Real y))))

(forall (x y)
  (if (buml:less-than x y)
      (and (not (buml:less-than y x))
           (not (= x y)))))

(forall (x y)
   (if (and (buml:Real x)
            (buml:Real y))
       (or (form:less-than x y)
           (form:less-than y x)
           (= y x))))

(forall (x y)
  (iff (buml:less-than x y)
       (and (buml:Real x)
            (buml:Real y)
            (not (buml:less-than y x))
            (not (= x y)))))

(forall (x y z)
  (if (and (buml:less-than x y)
           (buml:less-than y z))
      (buml:less-than x z)))

(forall (x z)
   (if (buml:less-than x z)
       (exists (y)
          (and (buml:less-than x y)
               (buml:less-than y z)))))
```

The add relation.

```
(forall (x y z)
   (if (buml:add x y z)
       (and (buml:Real x)
            (buml:Real y)
            (buml:Real z))))
```

```
(forall (x y)
   (if (and (buml:Real x)
            (buml:Real y))
       (exists (z)
          (buml:add x y z))))

(forall (x y z1 z2)
   (if (and (buml:add x y z1)
            (buml:add x y z2))
       (= z1 z2)))

(forall (x y zxy zyx)
   (if (and (buml:add x y zxy)
            (buml:add y x zyx))
       (= zxy zyx)))

(forall (x y z xy yz rxy ryz)
   (if (and (buml:add x y xy)
            (buml:add xy z rxy)
            (buml:add y z yz)
            (buml:add x yz ryz))
       (= rxy ryz)))

(forall (x y xz yz)
   (if (and (buml:add x z xz)
            (buml:add y z yz)
            (buml:less-than x y))
       (buml:less-than xz yz)))

(buml:Real  form:0)

(forall (x x0)
   (if (buml:add x form:0 x0)
       (= x x0)))

(forall (x y)
   (iff (buml:neg x y)
        (buml:add x y form:0)))

(forall (x)
  (if (buml:Real x)
      (exists (nx)
        (buml:neg x nx))))
```

The mult relation.

```
(forall (x y z)
   (if (buml:mult x y z)
       (and (buml:Real x)
            (buml:Real y)
            (buml:Real z))))

(forall (x y)
   (if (and (buml:Real x)
            (buml:Real y))
       (exists (z)
          (buml:mult x y z))))

(forall (x y z1 z2)
   (if (and (buml:mult x y z1)
            (buml:mult x y z2))
       (= z1 z2)))
```

```
(forall (x y zxy zyx)
   (if (and (buml:mult x y zxy)
            (buml:mult y x zyx))
       (= zxy zyx)))
```

```
(forall (x y z xy yz rxy ryz)
   (if (and (buml:mult x y xy)
            (buml:mult xy z rxy)
            (buml:mult y z yz)
            (buml:mult x yz ryz))
       (= rxy ryz)))

(forall (x y xy)
   (if (and (buml:mult x y xy)
            (buml:less-than form:0 x)
            (buml:less-than form:0 y))
       (buml:less-than form:0 xy)))
```

```
(buml:Real form:1)

(forall (x x1)
   (if (buml:mult x form:1 x1)
       (= x x1)))
```

```
(forall (x y)
   (iff (buml:inv x y)
        (buml:mult x y form:1)))

(forall (x)
  (if (and (buml:Real x)
           (not (= x form:0)))
      (exists (ix)
        (buml:inv x ix))))

(forall (x y z yz rxyz xy xz xyxz)
   (if (and (add y z yz)
            (mult x yz xyz)
            (mult x y xy)
            (mult x z xz)
            (add xy xz xyxz))
       (= xyz xyxz)))
```

Rational, integer, natural, unlimited natural, and whole numbers.

```
(forall (x)
  (if (form:RationalNumber x)
      (buml:Real x)))

(forall (x z)
   (if (and (form:Rational x)
            (form:Rational z)
            (buml:less-than x z))
       (exists (y)
          (and (form:Rational y)
               (buml:less-than x y)
               (buml:less-than y z)))))

(forall (x z)
   (if (and (form:Rational x)
            (form:Rational z)
            (buml:less-than x z))
       (exists (y)
          (and (buml:Real y)
               (not (form:Rational y))
               (buml:less-than x y)
               (buml:less-than y z)))))

(forall (y)
   (if (and (buml:Real y)
            (not (form:Rational y)))
       (and (not (exists (xu)
                   (and (form:Rational xu)
                        (forall (x)
                           (and (Rational x)
                                (buml:less-than x y)
                                (or (buml:less-than x xu)
                                   (= x xu)))))))
            (not (exists (zl)
                     (and (form:Rational zl)
                          (forall (z)
                            (and (Rational z)
                                 (buml:less-than y z)
                                 (or (buml:less-than zl z)
                                    (= zl z))))))))))
(forall (x)
  (if (buml:Integer x)
      (form:Rational x)))

(forall (x y)
  (iff (form:add-one x y)
       (form:add x form:1 y)))

(forall (x)
   (iff (buml:Integer x)
        (or (= x form:0)
            (exists (y)
               (and (buml:Integer y)
                    (or (form:add-one y x)
                        (form:add-one x y)))))))

(forall (x)
   (iff (form:NaturalNumber x)
        (and (buml:Integer x)
             (or (= x form:0)
                 (buml:less-than form:0 x)))))

(forall (x)
   (iff (buml:UnlimitedNatural x)
        (or (form:NaturalNumber x)
            (= x buml:*))))

(forall (x)
   (iff (form:WholeNumber x)
        (and (form:NaturalNumber x)
             (not (= x form:0)))))
```

10.3.1.3 Sequences

Sequences are a finite series of things, where the same thing can appear more than once in the series. The serial aspect is represented by positions, each of which identifies exactly one thing. This thing can be the same as ones identified by other positions in the series.

The in-sequence relation links sequences to their positions.

```
(forall (s pt)
   (if (form:in-sequence s pt)
       (and (form:Sequence s)
            (form:Position pt))))

(forall (s1 s2 pt)
   (if (and (form:in-sequence s1 pt)
            (form:in-sequence s2 pt))
```

```
        (= s1 s2)))

(forall (pt)
   (if (form:Position pt)
       (exists (s)
          (form:in-sequence s pt))))
```

The before-in-sequence relation serializes positions.

```
(forall (s pt1 pt2)
   (if (form:before-in-sequence s pt1 pt2)
       (and (buml:Sequence s)
            (form:Position pt1)
            (form:Position pt2))))

(forall (s pt1 pt2)
   (if (form:before-in-sequence s pt1 pt2)
       (and (form:in-sequence s pt1)
            (form:in-sequence s pt2))))

(not (exists (s pt1 pt2)
   (and (form:before-in-sequence s pt1 pt2)
        (form:before-in-sequence s pt2 pt1))))

(forall (s pt1 pt2 pt11 pt22)
   (if (and (form:before-in-sequence s pt1 pt2)
            (form:before-in-sequence s pt11 pt22))
       (iff (= pt1 pt11)
            (= pt2 pt22))))

(forall (s)
   (if (form:Sequence s)
       (if (exists (pt)
               (form:in-sequence s pt))
           (and (exists (pt1)
                   (not (exists (pt2)
                           (form:before-in-sequence s pt1 pt2))))
                (exists (pt2)
                   (not (exists (pt1)
                           (form:before-in-sequence s pt1 pt2))))))))
(forall (s pt1 pt11)
   (if (and (form:Sequence s)
            (not (exists (pt2)
                    (form:before-in-sequence s pt1 pt2)))
            (not (exists (pt2)
                    (form:before-in-sequence s pt11 pt2))))
       (= pt1 pt11)))

(forall (s)
   (if (form:Sequence s)
       (exists (pt2)
          (not (exists (pt1)
                  (form:before-in-sequence s pt1 pt2))))))

(forall (s pt2 pt22)
   (if (and (form:Sequence s)
            (not (exists (pt1)
                    (form:before-in-sequence s pt1 pt2)))
            (not (exists (pt1)
                    (form:before-in-sequence s pt1 pt22))))
        (= pt2 pt22)))
```

An empty sequence has no positions.

```
(forall (s)
   (if (form:empty-sequence s)
       (form:Sequence s)))

(forall (s)
   (if (form:Sequence s)
       (iff (form:empty-sequence s)
            (not (exists (pt)
                    (form:in-sequence s pt))))))
The position-count relation links positions to how far they are along in the sequence.
(forall (s pt n)
   (if (form:position-count s pt n)
       (and (form:Sequence s)
            (form:Position pt)
            (buml:UnlimtedNatural n))))

(forall (s pt n1 n2)
   (if (and (form:position-count s pt n1)
            (form:position-count s pt n2))
       (= n1 n2)))

(forall (s pt)
   (if (form:in-sequence s pt)
       (exists (n)
          (form:position-count s pt n))))

(forall (s pt2)
   (if (and (form:Sequence s)
            (not (exists (pt1)
                    (form:before-in-sequence s pt1 pt2))))
       (form:position-count s pt2 form:1)))

(forall (s pt1 n1 pt2 n2)
   (if (and (form:position-count s pt1 n1)
            (form:before-in-sequence s pt1 pt2)
            (form:position-count s pt2 n2))
       (form:add-one n1 n2)))
```

The sequence-length relation links sequences to how many positions they have.

```
(forall (s n)
    (if (form:sequence-length s n)
        (and (form:Sequence s)
             (buml:UnlimtedNatural n))))

(forall (s n)
   (iff (form:sequence-length s n)
        (or (and (form:empty-sequence s)
                 (= n form:0))
```

```
            (exists (pt1)
               (and (not (exists (pt2)
                            (form:before-in-sequence s pt1 pt2)))
                    (form:position-count s pt1 n))))))
```

The in-position relation links positions to things they identify. Each position identifies exactly one thing.

```
(forall (pt x)
    (if (form:in-position pt x)
         (form:Position pt)))

 (forall (pt x1 x2)
    (if (and (form:in-position pt x1)
             (form:in-position pt x2))
        (= x1 x2)))

 (forall (pt)
   (if (form:Position pt)
       (exists (x)
           (form:in-position pt x))))
```

The in-position-count relation links sequences to a thing based on how far along the position is in the sequence.

```
(forall (s n x)
   (if (form:in-position-count s n x)
       (and (buml:Sequence s)
            (form:NaturalNumber n))))

(forall (s n x)
   (iff (form:in-position-count s n x)
        (exists (pt)
           (and (form:in-position pt x)
                (form:position-count s pt n)))))
```

The same-sequence relation is true for sequences that identify the same things in the same order.

```
(forall (s1 s2)
   (if (form:same-sequence s1 s2)
       (and (form:Sequence s1)
            (form:Sequence s2))))

(forall (s1 s2)
   (iff (form:same-sequence s1 s2)
        (forall (x n)
           (iff (form:in-position-count s1 n x)
                (form:in-position-count s2 n x)))))
```

10.3.1.4 Strings

Strings are sequences of characters.

```
(forall (s)
    (if (buml:String s)
        (form:Sequence s)))

(forall (s pt x)
    (if (and (buml:String s)
             (form:in-sequence s pt)
             (form:in-position pt x))
        (form:Character x)))
```

The string-index-character relation links strings to a character based on how far along the position is in the sequence.

```
(forall (s n ch)
   (if (form:string-index-character s n ch)
       (and (buml:String s)
            (form:WholeNumber n)
            (form:Character ch))))

(forall (s n ch)
   (if (buml:String s)
       (iff (form:string-index-char s n ch)
            (form:in-position-count s n ch))))
The string-length relation links strings to how many positions they have.
(forall (s n)
   (if (form:string-length s n)
       (and (buml:String s)
            (form:NaturalNumber n))))

(forall (s)
   (if (buml:String s)
       (forall (n)
          (iff (form:string-length s n)
               (form:sequence-length s n)))))
```

The same-string relation is true for strings that identify the same characters in the same order.

```
(forall (s1 s2)
   (if (form:same-string s1 s2)
       (and (buml:String s1)
            (buml:String s2))))

(forall (s1 s2)
   (if (and (buml:String s1)
            (buml:String s2))
       (iff (form:same-string s1 s2)
            (form:same-sequence s1 s2))))
```

10.3.2 Classification and Generalization

Classification links classifiers to the things they classify. Classifiers are categories into which things fall.

```
(forall (c o f)
   (if (form:classifies c o f)
       (and (buml:Classifier c)
            (psl:state f))))
```

A classifier is more general than another when the things classified by the specialized classifier are classified by the general classifier.

```
(forall (csub csuper o f)
   (iff (buml:general csub csuper)
        (if (form:classifies csub o f)
            (form:classifies csuper o f))))
```

Classification applies in all PSL states prior to legal occurrences or to none. In PSL, states holding after an occurrence are the same as states prior to legal successor occurrences, so this constraint applies to states holding after occurrences as well as prior.

```
(forall (occ f c o)
   (if (and (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f occ)
            (form:classifies c o f))
       (forall (f2)
          (if (psl:prior f2 occ)
              (form:classifies c o f2)))))
```

10.3.3 Classifier Cardinality

Classifier cardinality is the number of things classified by a classifier in a PSL state.

```
(forall (c card f)
  (if (form:classifier-cardinality c card f)
      (and (buml:Classifier c)
           (form:NaturalNumber card)
           (psl:state f))))

(forall (c1 c2)
  (if (and (buml:Classifier c1)
           (buml:Classifier c2))
      (buml:Classifier (form:union c1 c2))))

(forall (c1 c2 o f)
  (iff (form:classifies (form:union c1 c2) o f)
       (or (form:classifies c1 o f)
           (form:classifies c2 o f))))

(forall (c1 c2 c1Card o1 f c1Card1)
   (if (and (buml:Classifer c1)
            (form:NaturalNumber c1Card)
            (form:classifer-cardinality c1 c1Card f)
            (not (form:classifies c1 o1 f))
            (buml:Classifier c2)
            (forall (o2)
               (iff (form:classifies c2 o2 f)
                    (= o1 o2)))
            (form:add-one c1Card c1Card1))
       (form:classifier-cardinality (form:union c1 c2) c1Card1 f)))
```

10.3.4 Properties

Properties link things to other things in a certain PSL state.

```
(forall (o p v f)
   (if (form:property-value o p v f)
       (and (buml:Property p)
            (psl:state f))))
```

Property values apply in all PSL states prior to legal occurrences or to none. In PSL, states holding after an occurrence are the same as states prior to legal successor occurrences, so this constraint applies to states holding after occurrences as well as prior.

```
(forall (occ f o p v)
   (if (and (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f occ)
            (form:property-value o p v f))
       (forall (f2)
          (if (psl:prior f2 occ)
              (form:property-value o p v f2)))))
```

Things with property values must be classified by the class owning the property (classes are classifiers in UML).

```
(forall (c p occ f o v)
   (if (and (buml:ownedAttribute c p)
            (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f occ)
            (form:property-value o p v f))
       (exists (f2)
          (and (psl:prior f2 occ)
               (form:classifies c o f2)))))
```

Property values must be classified by the type of the property.

```
(forall (p c occ f o v)
   (if (and (buml:type p c)
            (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f occ)
            (form:property-value o p v f))
       (exists (f2)
          (and (psl:prior f2 occ)
               (form:classifies c v f2)))))
```

The achieves-property-value relation links objects, properties, and values to occurrences that give the property the value. The property does not have the achieved value before the occurrence and does after.

```
(forall (o p v occ)
   (iff (form:achieves-property-value o p v occ)
        (and (forall (f)
                (if (form:priorA f occ)
                    (not (form:property-value o p v f))))
             (exists (f)
                (and (form:holdsA f occ)
                     (form:property-value o p v f))))))
```

Property value cardinality is number of values of a property on a particular thing.

```
(forall (o p n f)
   (if (form:property-value-cardinality o p n f)
       (and (buml:Property p)
             (psl:state f)
             (form:NaturalNumber n))))

(forall (occ f o p n)
   (if (and (psl:occurrence occ)
             (psl:legal occ)
             (psl:prior f occ)
             (form:property-value-cardinality o p n f))
       (exists (fv fp fvc vc)
          (and (psl:prior fv occ)
               (psl:prior fp occ)
               (psl:prior fvc occ)
               (forall (v)
                   (iff (form:classifies vc v fv)
                        (form:property-value o p v fp)))
               (form:classifier-cardinality vc n fvc)))))
```

Property value cardinality is constrained by the multiplicity of the property, but when the constraints are enforced is not defined in UML. For example, when an object is created, it will violate non-zero lower multiplicities on its properties.

```
(forall (p m c occ fc o n fp)
   (if (and (buml:lower p m)
             (buml:ownedAttribute c p)
             (psl:occurrence occ)
             (psl:legal occ)
             (psl:prior fc occ)
             (form:classifies c o fc)
             (psl:prior fp occ)
             (form:property-value-cardinality o p n fp))
       (or (= m n)
           (form:less-than m n))))

(forall (p m c occ fc o n fp)
   (if (and (buml:upper p m)
             (buml:ownedAttribute c p)
             (psl:occurrence occ)
             (psl:legal occ)
             (psl:prior fc occ)
             (form:classifies c o fc)
             (psl:prior fp occ)
             (form:property-value-cardinality o p n fp))
       (or (= m n)
           (form:less-than n m))))
```

Composite properties collectively do not have values that form cycles. Destruction propagation is not formalized because the execution model does not destroy things.

```
(forall (x y f)
   (iff (form:composite-link-trans x y f)
        (forall (occ)
           (if (and (psl:occurrence occ)
                    (psl:legal occ)
                    (psl:prior f occ))
               (or (exists (f2 p)
                        (and (psl:prior f2 occ)
                             (form:property-value y p x f2)
```

```
                          (buml:aggregation p buml:composite)))
                  (exists (f2 p z f3)
                     (and (psl:prior f2 occ)
                          (form:property-value z p x f2)
                          (buml:aggregation p buml:composite)
                          (psl:prior f3 occ)
                          (form:composite-link-trans z y f3))))))))
(forall (occ)
   (iff (and (psl:occurrence occ)
             (psl:legal occ))
        (not (exists (x f)
                 (and (psl:prior f occ)
                      (form:composite-link-trans x x f))))))
```

Properties in bUML are ordered and non-unique, which means the values are ordered, and the same value can appear more than once in the order. The property-value-sequence gives the sequence of values of a property in a PSL state.

```
(forall (o p s f)
   (if (form:property-value-sequence o p s f)
       (and (buml:Property p)
            (form:Sequence s)
            (psl:state f))))
```

Property values sequences apply in all PSL states prior to legal occurrences or to none. In PSL, states holding after an occurrence are the same as states prior to legal successor occurrences, so this constraint applies to states holding after occurrences as well as prior.

```
(forall (occ f o p s)
   (if (and (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f occ)
            (form:property-value-sequence o p s f))
       (forall (f2)
          (if (psl:prior f2 occ)
              (form:property-value-sequence o p s f2)))))

(forall (occ f1 f2 o p s1 s2)
   (if (and (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f1 occ)
            (psl:prior f2 occ)
            (form:property-value-sequence o p s1 f1)
            (form:property-value-sequence o p s2 f2))
       (= s1 s2)))

(forall (occ f)
    (if (and (psl:occurrence occ)
             (psl:legal occ)
             (psl:prior f occ))
        (forall (o p v)
           (iff (form:property-value o p v f)
                (exists (f2 s pt)
                   (and (psl:prior f2 occ)
                        (form:property-value-sequence o p s f2)
                        (form:in-sequence s pt)
                        (form:in-position pt v)))))))
```

```
(forall (occ f)
   (if (and (psl:occurrence occ)
            (psl:legal occ)
            (psl:prior f occ))
       (forall (o p)
          (iff (not (exists (v)
                          (form:property-value o p v f)))
               (or (not (exists (f2 s)
                               (and (psl:prior f2 occ)
                                    (form:property-value-sequence o p s f2))))
                   (exists (f2 s)
                      (and (psl:prior f2 occ)
                           (form:property-value-sequence o p s f2)
                           (form:empty-sequence s))))))))
```

10.4 Behavior

This subclause covers the behavioral aspects of the base semantics.

10.4.1 Property Value Modifiers

This subclause specifies PSL activities that modify property values.

```
(forall (o p v a)
   (if (or (form:add-property-value o p v a)
           (form:remove-property-value o p v a))
       (and (buml:Property p)
            (psl:activity a))))

(forall (o p v a aocc)
   (if (and (form:add-property-value o p v a)
            (psl:occurrence_of aocc a))
       (exists (f)
          (and (psl:holds f aocc)
               (form:property-value o p v f)))))

(forall (a o p v aocc)
   (if (and (form:remove-property-value o p v a)
            (psl:occurrence_of aocc a))
       (forall (f)
           (and (psl:holds f aocc)
                (not (form:property-value o p v f))))))

(forall (o p s a aocc)
   (if (and (form:set-property-value-sequence o p s a)
            (psl:occurrence_of aocc a))
       (exists (f s2)
          (and (psl:holds f aocc)
               (form:property-value-sequence o p s2 f)
               (form:same-sequence s s2)))))

(forall (o p a aocc)
   (if (and (form:clear-property-value-sequence o p a)
            (psl:occurrence_of aocc a))
```

```
        (exists (f s)
           (and (psl:holds f aocc)
                (form:property-value-sequence o p s f)
                (form:empty-sequence s)))))
```

10.4.2 Common Behavior

This subclause covers the semantics of elements used in all behaviors. It includes Operations, which appear in the bUML Kernel, rather than Common Behavior.

10.4.2.1 Syntax

Behaviors are classes of executions, as in UML. Behaviors specify constraints on their valid executions.

Operations are formalized as abstract behaviors that specify only inputs and outputs. More details about the executed behavior are not determined until runtime, when the operation is called on a particular object and a more detailed behavior is selected ("dispatch"). Operations are not formalized as features or properties on classes because they have no values at runtime. This subclause adds a generalization of bml:Behavior and buml:Operation that parameterizes them (form:ProcessDefinition).

```
(forall (pd)
   (if (form:ProcessDefinition pd)
       (and (buml:Class pd)
            (psl:activity pd))))

(forall (x)
   (if (or (buml:Behavior x)
           (buml:Operation x))
       (form:ProcessDefinition x)))

(forall (op b c)
   (if (form:method op b c)
       (and (buml:Operation op)
            (buml:Behavior b)
            (buml:Class c))))
```

Parameters are formalized as properties. The value of a parameter as a property on an execution is the value of the parameter for that execution in a particular PSL state. This assumes no inout parameters.

```
(forall (p)
   (if (buml:Parameter p)
       (buml:Property p)))

(forall (pd p)
  (if (form:ownedParameter pd p)
      (and (form:ProcessDefinition pd)
           (buml:Parameter p))))

(forall (po p)
  (if (buml:ownedParameter po p)
      (form:ownedParameter po p)))

(forall (po1 p po2)
   (if (and (form:ownedParameter po1 p)
            (form:ownedParameter po2 p))
        (= po1 po2)))

(forall (p)
    (if (buml:Parameter p)
        (exists (po)
           (form:ownedParameter po p))))

(forall (po p)
    (if (form:ownedParameter po p)
        (buml:ownedAttribute po p)))

(forall (p dk)
    (iff (form:InputParameter p)
         (and (buml:direction p dk)
              (= dk buml:in))))

(forall (p)
    (iff (form:OutputParameter p)
         (forall (dk)
            (and (buml:direction p dk)
                 (or (= dk buml:out)
                     (= dk buml:return))))))
```

10.4.2.2 Semantics

Behaviors are classes of executions. Behaviors specify constraints on their valid executions. Executions are interpreted as PSL activity occurrences, which represent one of potentially many possible traces that might transpire when the execution model is executing. Behaviors classify their executions independently of PSL state (the classifies relation on behaviors is never used with PSL states that are constrained against occurrences), and similarly for property values of occurrences when the values are also occurrences.

```
(forall (pd x f)
    (if (and (form:ProcessDefinition pd)
             (form:classifies pd x f))
        (form:execution x)))

(forall (pd x f)
    (if (and (form:ProcessDefinition pd)
             (form:classifies pd x f))
        (forall (f2)
           (form:classifies pd x f2))))
```

The rest of the axioms in this subclause relate PSL occurrences to executions to support multiple classification of executions.

```
(forall (x occ)
    (if (form:execution-occ x occ)
        (and (form:execution x)
             (psl:activity_occurrence occ))))

(forall (x1 x2 occ)
    (if (and (form:execution-occ x1 occ)
             (form:execution-occ x2 occ))
        (= x1 x2)))
```

```
(forall (x occ1 occ2)
   (if (and (form:execution-occ x occ1)
            (form:execution-occ x occ2))
       (form:same-suboccs occ1 occ2)))

(forall (occ1 occ2 subocc)
   (iff (form:same-suboccs occ1 occ2)
        (iff (form:subactivity-occurrence-neq subocc occ1)
             (form:subactivity-occurrence-neq subocc occ2))))

(forall (x)
   (if (form:execution x)
       (and (psl:activity_occurrence x)
            (exists (xocc)
               (and (form:execution-occ x xocc)
                    (= x xocc))))))

(forall (pd x f)
   (if (and (form:ProcessDefinition pd)
            (form:classifies pd x f)
            (not (psl:atomic pd)))
       (exists (occ)
          (and (form:execution-occ x occ)
               (psl:occurrence_of occ pd)))))

(forall (pd x f)
   (if (and (form:ProcessDefinition pd)
            (form:classifies pd x f)
            (psl:atomic pd))
       (exists (occ cab)
          (and (form:execution-occ x occ)
               (form:complex-atomic cab pd)
               (psl:occurrence_of occ cab)))))

(forall (b cab)
   (iff (form:complex-atomic cab b)
        (and (not (psl:atomic cab))
             (atomic b)
             (forall (cabocc)
                (if (psl:occurrence_of cabocc cab)
                    (exists (bocc)

                       (and (psl:occurrence_of bocc b)
                            (psl:root_occ bocc cabocc)
                            (psl:leaf_occ bocc cabocc))))))))
```

10.4.3 Activity Edges Generally

This subclause specifies additional syntactic relations on activity edges for the formalization, including a generalization of activity edges that generalizes activity nodes also (form:ActivityElement), see 10.4.4.

```
(forall (x)
   (if (buml:ActivityEdge x)
       (form:ActivityElement x)))
```

```
(forall (n)
  (if (buml:ActivityNode n)
      (iff (form:max-one-incoming-edge-node n)
           (forall (e1 e2)
               (if (and (buml:incoming n e1)
                        (buml:incoming n e2))
                   (= e1 e2))))))

(forall (n)
  (iff (form:no-incoming-edge n)
       (not (exists (e)
                (buml:incoming n e)))))

(forall (n)
  (iff (form:no-outgoing-edge n)
       (not (exists (e)
                (buml:outgoing n e)))))
(forall (n)
  (iff (form:max-one-incoming-edge n)
       (forall (e1 e2)
           (if (and (buml:incoming n e1)
                    (buml:incoming n e2))
               (= e1 e2)))))

(forall (n)
   (iff (form:max-one-outgoing-edge n)
        (forall (e1 e2)
           (if (and (buml:outgoing n e1)
                    (buml:outgoing n e2))
               (= e1 e2)))))
```

10.4.4 Activity Nodes Generally

10.4.4.1 Syntax

This subclause specifies additional syntactic relations on activity edges for the formalization, including a generalization of activity nodes that generalizes activity edges also (form:ActivityElement), see 10.4.3.

```
(forall (x)
   (if (buml:ActivityNode x)
       (form:ActivityElement x)))
```

Executable nodes and object nodes are formalized as properties (executable nodes generalize actions and structured nodes, and object nodes generalize pins and activity parameter nodes). Execution nodes are properties of the activities that contain them, typed by behaviors that vary according by the particular executable node. The value of an execution node as a property on an activity execution is the execution of that node. The value of an object node as a property on an activity execution is the value in that object node in a particular PSL state.

```
(forall (n)
   (if (or (buml:ExecutableNode n)
           (buml:ObjectNode n))
       (buml:Property n)))
```

The activity relation in the formalization links activity elements to the activity that contains them, regardless of intervening structured nodes.

```
(forall (ae a)
  (if (form:activity ae a)
      (and (form:ActivityElement ae)
           (buml:Activity a))))

(forall (ae a1 a2)
   (if (and (form:activity ae a1)
            (form:activity ae a2))
       (= a1 a2)))

(forall (ae)
  (if (form:ActivityElement ae)
      (exists (a)
         (form:activity ae a))))

(forall (ae a)
  (if (buml:activity ae a)
      (form:activity ae a)))

(forall (n a)
   (if (and (or (buml:ExecutableNode n)
                (buml:ObjectNode n))
            (form:activity n a))
       (buml:ownedAttribute a n)))
```

Executable nodes as properties have exactly one type, which is a behavior (UML allows at most one type per property).

```
(forall (n b)
   (if (and (buml:ExecutableNode n)
            (buml:type n b))
       (buml:Behavior b)))

(forall (n)
   (if (buml:ExecutableNode n)
       (exists (b)
          (buml:type n b))))
```

10.4.4.2 Semantics

Executions that are values of executable nodes as properties of activity executions are PSL subactivity occurrences of the activity execution. PSL subactivity occurrences happen during their superoccurrences.

```
(forall (n a xa f xn)
   (if (and (buml:ExecutableNode n)
            (form:activity n a)
            (form:classifies a xa f)
            (form:property-value xa n xn f))
       (form:subactivity_occurrence-neq xn xa)))
```

Executions that are values of executable nodes as properties of activity executions are values in all PSL states or none. PSL states are not used to formalize the state of execution.

```
(forall (n a xa f xn)
   (if (and (buml:ExecutableNode n)
            (form:activity n a)
            (form:classifies a xa f)
            (form:property-value xa n xn f))
       (forall (f2)
           (form:property-value xa n xn f2))))

 (forall (n a xa1 xa2 f xn rxa1 rxa2)
   (if (and (buml:ExecutableNode n)
            (form:activity n a)
            (form:classifies a xa1 f)
            (form:classifies a xa2 f)
            (form:property-value xa1 n xn f)
            (form:property-value xa2 n xn f)
            (psl:root_occ rxa1 xa1)
            (psl:root_occ rxa2 xa2))
       (= rxa1 rxa2)))
```

10.4.5 Structured Nodes Generally

Executions that are values of structured nodes as properties of activity executions are PSL subactivity occurrences of the structured node execution. PSL subactivity occurrences happen during their superoccurrences.

```
(forall (n sn a xa f xn xsn)
   (if (and (buml:inStructuredNode n sn)
            (buml:ExecutableNode n)
            (form:activity n a)
            (form:classifies a xa f)
            (form:property-value xa n xn f)
            (form:property-value xa sn xsn f))
       (form:subactivity_occurrence-neq xn xsn)))

 (forall (ip a)
  (if (and (buml:InputPin ip)
           (form:activity ip a))
      (iff (form:required-inputpin ip)
           (forall (ipmin)
              (if (buml:lower ip ipmin)
                  (not (= ipmin form:0)))))))
(forall (n)
   (iff (form:executable-without-input n)
        (and (buml:ExecutableNode n)
             (form:no-incoming-edge n)
             (forall (ip)

                 (if (buml:input n ip)
                     (not (or (form:required-inputpin ip)
                              (exists (e)
                                 (buml:incoming ip e)))))))))
```

This constraint applies to the syntactic pattern of a structured activity node containing executable nodes that have no incoming control flows, no incoming object flows to any pins, and no required inputs. It requires the contained nodes to execute when the structured node does.

```
(forall (sn n a)
   (if (and (buml:inStructuredNode sn n)
            (form:executable-without-input n)
            (form:activity n a))
```

```
        (forall (xa f xsn)
             (if (and (form:classifies a xa f)
                      (form:property-value xa sn xsn f))
                 (exists (xn)
                    (form:property-value xa n xn f))))))

(forall (a xa sn xsn)
 (iff (form:move-structured-pin-values a xa sn xsn)
      (and (forall (ip on2 fxsn sip xsnroot asnroot)
             (if (and (buml:input sn ip)
                      (form:structured-input-or-output ip on2)
                      (form:flows-trans-fork-merge ip on2)
                      (form:priorA fxsn xsn)
                      (form:property-value-sequence xa ip sip fxsn)
                      (psl:root_occ xsnroot xsn)
                      (psl:occurrence_of xsnroot asnroot))
                 (form:set-property-value-sequence xa on2 sip asnroot)))
           (forall (op on2 xsnleaf asnleaf fxsnleaf son2)
             (if (and (buml:output sn op)
                      (form:structured-input-or-output on2 op)
                      (form:flows-trans-fork-merge on2 op)
                      (psl:leaf_occ xsnleaf xsn)
                      (psl:occurrence_of xsnleaf asnleaf)
                      (form:priorA fxsnleaf xsnleaf)
                      (form:property-value-sequence xa on2 son2 fxsnleaf))
                 (and (form:clear-property-value-sequence xa on2 asnleaf)
                      (form:set-property-value-sequence xa op son2 asnleaf)))))))
```

This constraint requires values of structured node pins to be transferred in and out of the structured node when the node execution begins and ends respectively.

```
(forall (sn a)
   (if (and (buml:StructuredNode sn)
            (form:activity sn a))
        (forall (xa f xsn)
             (if (and (form:classifies a xa f)
                      (form:property-value xa sn xsn f))
                 (form:move-structured-pin-values a xa sn xsn)))))
```

This constraint ensures executions of isolated structured nodes do not read objects modified by external executions during the execution of the structured node.

```
(forall (sn a)
   (if (and (buml:StructuredNode sn)
            (buml:isMustIsolate sn form:true)
            (form:activity sn a))
        (forall (xa f xsn xacrsf xa2 acrsf oip o fxacrsf)
           (if (and (form:classifies a xa f)
                    (form:property-value xa sn xsn f)
                    (form:subactivity_occurrence-neq xacrsf xsn)
                    (psl:subactivity_occurrence xa2 xsn)
                    (form:property-value xa2 acrsf xacrsf f)
                    (buml:ReadStructuralFeatureAction acrsf)
                    (buml:object acrsf oip)
                    (form:priorA xacrsf fxacrsf)
                    (form:property-value xa2 oip o fxacrsf))
```

```
              (not (exists (xout p v)
                      (and (not (form:subactivity_occurrence-neq xout xsn))
                           (form:achieves-property-value o p v xout))))))))
```

10.4.6 Expansion Regions

10.4.6.1 Syntax

Expansion regions are formalized as call actions, where the activity called is constructed from the nodes in the region. The activity has parameter nodes corresponding to the pins of the expansion region, including the expansion nodes.

```
(forall (n)
   (if (buml:ExpansionRegion n)
       (buml:CallAction n)))
```

The expansion-activity relation links expansion regions to the constructed activity it calls.

```
(forall (ac a)
   (if (form:expansion-activity ac a)
       (and (buml:ExpansionRegion ac)
            (buml:Activity a))))

(forall (ac a)
   (if (form:expansion-activity ac a)
       (form:called ac a)))

(forall (ac1 ac2 a)
   (if (and (form:expansion-activity ac1 a)
            (form:expansion-activity ac2 a))
       (= ac1 ac2)))
Expansion nodes are formalized as pins.
(forall (n)
   (if (buml:ExpansionNode n)
       (buml:Pin n)))
```

10.4.6.2 Semantics

All the values of input expansion nodes are taken as a single collection, whereas in UML each value is taken as a separate collection. It is assumed edges do not cross expansion region boundaries and expansion regions are not nested.

The expansion-input-value-xcall relation links executions of expansion region actions (the xac variable) and positions (pt) in a property value sequence of an input expansion node with executions of its constructed activity (xcall) (in the execution model, all input expansion nodes have the same number of values at the beginning of region execution). For each expansion region action executions, this relation is one-to-one between positions and executions of the constructed activity, see the necessary condition on expansion region executions at the end of this subclause.

```
(forall (xac pt xcall)
   (if (form:expansion-input-value-xcall xac pt xcall)
       (and (form:execution xac)
            (form:Position pt)
            (form:execution xcall))))
(forall (xac pt1 xcall1 pt2 xcall2)
   (if (and (form:expansion-input-value-xcall xac pt1 xcall1)
            (form:expansion-input-value-xcall xac pt2 xcall2))
       (iff (= pt1 pt2)
            (= xcall1 xcall2))))
```

The expanded-value-to-fill relation links expansion region actions (the xac variable) under activity executions (xa), executions of the expansion region constructed activity (xcall) to input expansion nodes of the region (ip) and a value of the expansion node (v) to pass to a constructed activity execution. It uses expansion-input-value-xcall to link each value of the expansion node to one of the constructed activity executions.

```
(forall (xa xac xcall ip v)
    (iff (form:expanded-value-to-fill xa xac xcall ip v)
         (forall (ptindex sindex n)
            (if (and (form:expansion-input-value-xcall xac ptindex xcall)
                     (form:position-count sindex ptindex n))
                (exists (fxac s pt)
                   (and (form:priorA fxac xac)
                        (form:property-value-sequence xa ip s fxac)
                        (form:position-count s pt n)
                        (form:in-position pt v)))))))
```

The fill-input-parameter-node relation ensures executions of expansion region actions (the xac variable) transfer values from an input pin of the expansion region (ip) to the corresponding input parameter nodes of the executions of the constructed activity (xcall). It assumes input pins never have more tokens than the action can accept in one execution, and input pin multiplicity upper is one or unlimited (ipmax).

```
(forall (xa xac xcall ip ipmax)
    (iff (form:fill-input-parameter-node xa xac xcall ip ipmax)
         (forall (srootxac arootxac p pnode a)
            (if (and (psl:root_occ srootxac xac)
                     (psl:occurrence_of srootxac arootxac)
                     (form:pin-parameter-match ip p)
                     (buml:parameter pnode p)
                     (form:activity pnode  a)
                     (form:activity ip a))
                (or (and (= ipmax form:1)
                         (forall (v)
                            (if (and (if (buml:ExpansionNode ip)
                                         (form:expanded-value-to-fill xa xac xcall ip v))
                                     (if (not (buml:ExpansionNode ip))
                                        (exists (fxac)
                                           (and (form:priorA fxac xac)
                                                (form:property-value xa ip v fxac)))))
                                 (form:add-property-value xcall pnode v arootxac))))
                    (and (= ipmax buml:*)
                         (forall (fxac s)
                            (if (and (form:priorA fxac xac)
                                     (form:property-value-sequence xa ip s fxac))
                                (form:set-property-value-sequence xcall pnode s
                                                                   arootxac)))))))))
```

The empty-output-parameter-node relation ensures executions of expansion region action (the xac variable) transfer values from an output parameter node of the executions of the constructed activity (xcall) to the corresponding output pin of the expansion region (op). It assumes output pin multiplicity upper (opmax) is one or unlimited.

```
(forall (xa xac xcall op opmax)
    (iff (form:empty-output-parameter-node xa xac xcall op opmax)
         (forall (sleafxac aleafxac p pnode a fxcall)
            (if (and (psl:leaf_occ sleafxac xac)
                  (psl:occurrence_of sleafxac aleafxac)
                  (form:pin-parameter-match op p)
                  (buml:parameter pnode p)
                  (form:activity pnode a)
                  (form:activity op a)
                  (form:holdsA fxcall xcall))
             (or (and (= opmax form:1)
                      (forall (v)
                          (if (form:property-value xcall pnode v fxcall)
                              (form:add-property-value xa op v aleafxac))))




                 (and (= opmax buml:*)
                      (forall (s)
                          (if (form:property-value-sequence xcall pnode s fxcall)
                              (form:set-property-value-sequence xa op s
                                                                aleafxac)))))))))
```

The fill-empty-parameter-node relation combines the fill-input-parameter-node relation and empty-output-parameter-node relations.

```
(forall (a xa ac xac xcall)
    (iff (form:fill-empty-parameter-node a xa ac xac xcall)
         (and (forall (ip ipmax)
                   (if (and (buml:input ac ip)
                            (not (buml:ExpansionNode ip))
                            (buml:upper ip ipmax))
                       (form:fill-input-parameter-node xa xac xac ip ipmax)))
              (forall (op opmax)
                   (if (and (buml:output ac op)
                            (not (buml:ExpansionNode op))
                            (buml:upper op opmax))
                    (form:empty-output-parameter-node xa xac xac op opmax))))))
```

The expansion-input-value-output relation links two sequences and their positions. The relation is one-to-one, except not all positions in the first sequence are necessarily in the second sequence. The relation preserves the ordering of the sequences.

```
(forall (s1 pt1 s2 pt2)
    (if (form:expansion-input-value-output s1 pt1 s2 pt2)
        (and (form:Sequence s1)
             (form:Position pt1)
             (form:in-sequence s1 pt1)
             (form:Sequence s2)
             (form:Position pt2)
             (form:in-sequence s2 pt2))))
(forall (s1 pt1 pt12 s2 pt21 pt22)
    (if (and (form:expansion-input-value-output s1 pt1 s2 pt12)
             (form:expansion-input-value-output s1 pt21 s2 pt22))
        (iff (= pt1 pt21)
             (= pt12 pt22))))
(forall (s1 pt1 pt12 s2 pt2 pt22 )
    (if (and (form:expansion-input-value-output s1 pt1 s2 pt2)
             (form:expansion-input-value-output s1 pt12 s2 pt22))
        (iff (forall (n1 n2)
              (and (form:position-count s1 pt1 n1)
                   (form:position-count s1 pt12 n2)
                   (form:less-than n1 n2)))
          (forall (n1 n2)
              (and (form:position-count s2 pt2 n1)
                   (form:position-count s2 pt22 n2)
                   (form:less-than n1 n2))))))
```

The contracted-sequence-to-empty relation links executions of expansion region actions (the xac variable) and executions of its constructed activity (xcall) with property value sequences (s) of an output pin (op). It assumes each execution of the body of the expansion region supplies no more than one value to each output expansion node. It ensures the value of the output parameter node of the constructed activity execution, if any, is in the output expansion node in the proper order. It uses the expansion-input-value-output relation for output value ordering.

```
(forall (xa xac xcall op sout)
    (iff (form:contracted-sequence-to-empty xa xac xcall op sout)
         (forall (ptin sin n pnode p a)
             (if (and (form:expansion-input-value-xcall xac ptin xcall)
                      (form:position-count sin ptin n)
                      (form:pin-parameter-match op p)
                      (buml:parameter pnode p)
                      (form:activity pnode  a)
                      (form:activity op a))
                 (and (forall (v fxcallv)
                          (if (and (form:holdsA fxcallv xcall)
                                   (form:property-value xcall pnode v fxcallv))
                              (exists (ptout)
                                  (and (form:in-sequence sout ptout)
                                       (form:in-position ptout v)
                                       (form:expansion-input-value-output sin ptin sout
                                                                          ptout)))))
                      (if (not (exists (v fxcallv)
                                   (and (form:holdsA fxcallv xcall)
                                        (form:property-value xcall pnode v fxcallv))))
                          (not (exists (ptout)
                                   (form:expansion-input-value-output sin ptin sout ptout))))
                      (forall (ptout)
                          (iff (form:in-sequence sout ptout)
                               (exists (ptin2)
                                   (form:expansion-input-value-output sin ptin2 sout
                                                                      ptout)))))))))
```

This is a necessary condition on executions of expansion region actions that its constructed activity execute as many times as there are input values in the input expansion nodes (in the execution model, all input expansion nodes have the same number of values at the beginning of region execution), that values are transferred between pins of the action and parameter nodes of the constructed activity execution, and if the region mode is iterative that the executions are ordered in time in the same way as the input values (see the three conditions in the large conjunction). This assumes parallel expansion nodes have no output expansion nodes.

```
(forall (ac a acall)
       (if (and (buml:ExpansionRegion ac)
                (form:activity ac a)
                (form:expansion-activity ac acall))
       (forall (xa xac f)
           (if (and (form:classifies a xa f)
            (form:property-value xa ac xac f))
      (and (forall (ipindex fxac s pt)
              (if (and (buml:input ac ipindex)
                         (buml:ExpansionNode ipindex)
                         (form:priorA fxac xac)
                          (form:property-value-sequence xa ipindex s fxac)
                         (form:in-sequence s pt))
                   (exists (xcall)
                      (and (form:classifies acall xcall f)
                            (form:subactivity_occurrence-neq xcall xac)
                             (form:expansion-input-value-xcall xac pt xcall)))))
            (forall (ptany xcall)
              (if (form:expansion-node-value-xcall xac ptany xcall)
                   (and (form:fill-empty-parameter-node a xa ac xac xcall)
                        (forall (ip)
                          (if (and (buml:input ac ip)
                                     (buml:ExpansionNode ip))
                               (form:fill-input-parameter-node xa xac xcall ip form:1)))
                        (forall (op)
                          (if (and (buml:output ac op)
                                     (buml:ExpansionNode op))
                              (and (form:empty-output-parameter-node xa xac xcall op
                                                                      form:1)
                                   (exists (s2 fxacs)
                                         (and (form:contracted-sequence-to-empty xa xac
                                                                              xcall op s2)
                                              (form:holdsA fxacs xac)
                                               (form:property-value-sequence xa op s2
                                                                              fxacs)))))))))
            (forall (ptany1 xcall1 ptany2 xcall2)
              (if (and (buml:mode ac buml:iterative)
                        (form:expansion-node-value-xcall xac ptany1 xcall2)
                        (form:expansion-node-value-xcall xac ptany2 xcall2)
                       (not (= xcall1 xcall2)))
                   (forall (xcall1root xcall1leaf xcall2root xcall2leaf)
                     (if (and (form:expansion-input-value-xcall xac ptany1 xcall2)
                                (form:expansion-input-value-xcall xac ptany2 xcall2)
                                (not (= xcall1 xcall2))
                                (psl:root_occ xcall1root xcall1)
                                (psl:leaf_occ xcall1leaf xcall1)
                                (psl:root_occ xcall2root xcall2)
                                (psl:leaf_occ xcall2leaf xcall2))
                         (or (psl:earlier xcall1root xcall2leaf)
                             (psl:earlier xcall1leaf xcall2root))))))))))
```

10.4.7 Control Flow

This subclause gives sufficient conditions for existence of action execution due to control flow, except for 10.4.7.3, which gives a necessary condition. The syntactic patterns of this subclause do not include any object flows.

10.4.7.1 Top level action

This subclause applies to the syntactic pattern of an action directly contained in an activity that requires no input to start. The constraint requires the action to execute for every execution of the containing activity.

```
(forall (n a)
   (if (and (buml:activity n a)
            (form:executable-without-input n))
       (forall (xa f)
            (if (form:classifies a xa f)
                (exists (xn)
                   (form:property-value xa n xn f))))))
```

10.4.7.2 Initial Node to Action

This subclause applies to the syntactic pattern of a control flow from an initial node to an action, with no more than one edge going out of the initial node, no more than one edge coming into the action, and no pins on the action (for example, see Statement Sequence pattern in A.4.1). It requires the action to execute for every execution of the containing activity.

```
(forall (n1 n2)
   (iff (form:same-syntactic-container n1 n2)
        (exists (c)
           (or (and (buml:inStructuredNode n1 c)
                    (buml:inStructuredNode n2 c))
               (and (buml:activity n1 c)
                    (buml:activity n2 c))))))

(forall (i e ac a)
   (if (and (buml:InitialNode i)
            (form:max-one-outgoing-edge i)
            (buml:target e ac)
            (form:same-syntactic-container i ac)
            (buml:Action ac)
            (not (exists (ip)
                      (buml:input ac ip)))
            (form:activity i a))
       (forall (xa sn xsn f)
          (if (and (form:classifies a xa f)
                   (or (not (buml:inStructuredNode i sn))
                       (form:property-value xa sn xsn f)))
              (exists (xac)
                 (form:property-value xa ac xac f))))))
```

10.4.7.3 Action to Action, general necessary condition

This subclause applies to syntactic pattern of a control flow between actions, with any intervening and chained control nodes, and regardless of any other flows. It requires each target action execution to follow no more than one source action execution, and each source action execution to be followed by no more than one source action execution. It does not require the target action to execute.

```
(forall (n1 n2)
   (iff (form:flow-trans-control-node n1 n2)
        (exists (e)
           (and (buml:outgoing n1 e)
                (or (buml:target e n2)
                    (exists (nt)
                       (and (buml:target e nt)
                            (buml:ControlNode nt)
                            (form:flow-trans-control-node nt n2))))))))
```

The follows relation links two PSL occurrences (s1 and s2 variables) under an execution of an activity (a), where the first occurrence happens sometime before the second. The follows relation is used in sufficient conditions for the existence of action executions due to control flow from other actions, for example in 10.4.7.4.

```
(forall (s1 s2 a)
   (if (form:follows s1 s2 a)
       (form:min-precedesA s1 s2 a)))

(forall (ac1 ac2 a)
   (if (and (buml:Action ac1)
            (buml:Action ac2)
            (form:flow-trans-control-node ac1 ac2)
            (form:activity ac1 a))
       (forall (xa f xac1 xac2 xac12 xac22)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac1 xac1 f)
                   (form:property-value xa ac2 xac2 f)
                   (form:property-value xa ac1 xac12 f)
                   (form:property-value xa ac2 xac22 f)
                   (form:follows xac1 xac2 a)
                   (form:follows xac12 xac22 a))
              (iff (= xac1 xac12)
                   (= xac2 xac22))))))
```

10.4.7.4 Action to Action, single control flow, optional merge/fork

This subclause applies to the syntactic pattern of a control flow between actions, with any intervening and chained fork and merge nodes (for example, see the Statement Sequence pattern in A.4.1). The target action has no other incoming control flows and no input pins. The constraint requires the target action to execute after the source action does.

The flow-trans-fork-merge links activity nodes that have a control or object flow between them (the n1 variable as source, n2 as target), possibly with intervening and chained fork and merge nodes. The flow will be a control flow if the nodes are actions and an object flow if the nodes are object nodes.

```
(forall (n1 n2)
   (iff (form:flow-trans-fork-merge n1 n2)
        (exists (e)
           (and (buml:outgoing n1 e)
                (or (buml:target e n2)
                    (exists (nt)
                       (and (buml:target e nt)
                            (or (buml:ForkNode nt)
                                (buml:MergeNode nt))
                            (form:flow-trans-fork-merge nt n2))))))))
(forall (ac1 ac2 a)
   (if (and (buml:Action ac1)
            (buml:Action ac2)
            (form:max-one-incoming-edge ac2)
            (form:flow-trans-fork-merge ac1 ac2)
            (not (exists (ip)
                   (buml:input ac ip)))
            (form:activity ac1 a))
       (forall (xa f xac1)
           (if (and (form:classifies a xa f)
                    (form:property-value xa ac1 xac1 f))
                (exists (xac2)
                   (and (form:property-value xa ac2 xac2 f)
                        (form:follows xac1 xac2 a)))))))
```

10.4.8 Object Flow

This subclause gives sufficient conditions for the presence of values in object nodes in 10.4.8.1 and 10.4.8.2 and for the existence of action execution due to object flow in 10.4.8.3 through 10.4.8.6, and also control flow in 10.4.8.4 through 10.4.8.6. The syntactic patterns of this subclause do not have object nodes with more than one outgoing edge (no token competition).

10.4.8.1 Object node to object node, optional fork/merge

This subclause applies to the syntactic pattern of object flow between object nodes, with any intervening and chained fork and merge nodes, where the flow is not into or out of a structured node pin (for example, see the two Instance Variable Assignment patterns in A.4.4 and A.4.5). The source object node has exactly one outgoing object flow. The constraints require values in the source object node be transferred to the target object node.

```
(forall (n sn)
   (iff (form:inStructuredNode-trans n sn)
        (or (buml:inStructuredNode n sn)
            (exists (nt)
               (and (buml:inStructuredNode n nt)
                    (form:inStructuredNode-trans nt sn))))))
```

The structured-input-or-output relation links input pins (the on1 variable) or output pins (on2) of structured nodes with object nodes in the structured node.

```
(forall (on1 on2)
   (iff (form:structured-input-or-output on1 on2)
        (exists (sn)
           (and (buml:StructuredNode sn)
                (or (and (buml:input sn on1)
                         (form:inStructuredNode-trans on2 sn))
                    (and (buml:output sn on2)
                         (form:inStructuredNode-trans on1 sn)))))))
(forall (on1 on2 a)
   (if (and (buml:ObjectNode on1)
            (buml:ObjectNode on2)
            (not (form:structured-input-or-output on1 on2))
            (form:max-one-outgoing-edge on1)
            (form:flows-trans-fork-merge on1 on2)
            (form:activity on1 a))
       (forall (xa f xsub v fon1s son1)
          (if (and (form:classifies a xa f)
                   (form:subactivity_occurrence-neq xsub xa)
                   (form:achieves-property-value xa son1 v xsub)
                   (psl:holds fon1s xsub)
                   (form:property-value-sequence xa on1 son1 fon1s))
              (exists (amove)
                 (and (form:clear-property-value-sequence xa on1 amove)
                      (form:set-property-value-sequence xa on2 son1 amove)
                      (form:subactivity_occurrence-neq
                         (psl:successor amove xsub) xa)))))))
```

10.4.8.2 Object node to object node, decision, optional fork/merge

This subclause applies to the syntactic pattern of object flow between object nodes, with any intervening and chained fork and merge nodes, one intervening decision node, where the flow is not into or out of a structured node pin (for example, see the Do-While Loop pattern in A.4.10). The decision input flow comes from an output pin on the same action as the output pin providing the decision input. The source object node has exactly one outgoing object flow. The constraints require the values in the source object node to be transferred to the target object node if the decision input matches the guard.

The flow-trans-fork-merge-decision relation links activity nodes that have a control or object flow between them (the n1 variable as source, n2 as target), where the flow is not into or out of a structured node pin, possibly with intervening and chained fork and merge nodes. The flow has one intervening decision node with a decision input flow from an output pin (dip), possibly with intervening and chained fork and merge nodes, and a guard specification (g). The flow will be a control flow if the nodes (n1 and n2) are actions and an object flow if the nodes are object nodes.

```
(forall (n1 n2 dip g)
    (iff (form:flow-trans-fork-merge-decision n1 n2 dip g)
         (exists (dn)
            (and (buml:DecisionNode dn)
                 (form:flow-trans-fork-merge n1 dn)
                 (form:flow-trans-fork-merge dn n2)
                 (exists (edn nt gvs)
                    (and (buml:outgoing dn edn)
                         (buml:target edn nt)
                         (or (= nt n2)
                             (form:flow-trans-fork-merge nt n2))
                             (buml:guard edn gvs)
                             (buml:value gvs g)))
                 (form:flow-trans-fork-merge dip dn)
                 (buml:OutputPin dip)
                 (exists (edn nt)
                    (and (buml:incoming dn edn)
                         (buml:decisionNodeInputFlow dn edn)
                         (buml:source edn nt)
                         (or (= nt dip)
                             (form:flow-trans-fork-merge dip nt))))))))
```

This assumes the output pin providing the decision input has exactly one value.

```
(forall (on1 on2 ac dip g a)
    (if (and (buml:ObjectNode on1)
             (buml:ObjectNode on2)
             (not (form:structured-input-or-output on1 on2))
             (buml:Action ac)
             (buml:output ac on1)
             (buml:output ac dip)
             (form:max-one-outgoing-edge on1)
             (form:max-one-outgoing-edge dip)
             (form:flow-trans-fork-merge-decision on1 on2 dip g)
             (form:activity on1 a))
        (forall (xa f xac fxac vdip son1)
           (if (and (form:classifies a xa f)
                    (form:property-value xa ac xac f)
                    (form:holdsA fxac xac)
                    (form:property-value xa dip vdip fxac)
                    (= vdip g)
                    (form:property-value-sequence xa on1 son1 fxac))
               (exists (amove occmove)
                  (and (form:remove-property-value xa dip vdip amove)
                       (form:clear-property-value-sequence xa on1 amove)
                       (form:set-property-value-sequence xa on2 son1 amove)
                       (psl:occurrence_of occmove amove)
                       (psl:next_subocc xac occmove a)))))))
```

10.4.8.3 Action with pins, no incoming control flow or one from initial

This subclause applies to the syntactic pattern of an action with pins, and no incoming control flow or one from an initial node (for example, see the Testing String Equality pattern in A.5.9). It requires the action to execute for every execution of the activity when the input pins are provided enough values to meet their lower multiplicity.

The action-input-pins-satisfied links actions (the ac variable) under executions of their containing activities (xa), with other executions (xsub) after which the values in the action's pins meet their lower multiplicity. An incoming control flow is required if all input pins are optional (lower multiplicity of zero). It assumes that the input pin multiplicity lower bound is zero or one.

```
(forall (ac xa xsub)
   (iff (form:action-input-pins-satisfied ac xa xsub)
        (and (forall (ip ipmin)
                (if (and (buml:input ac ip)
                         (buml:lower ip ipmin))
                    (or (= ipmin form:0)
                        (and (= ipmin form:1)
                             (exists (v f)
                                (and (form:holdsA f xsub)
                                     (form:property-value xa ip v f)))))))
             (or (exists (e)
                    (buml:target e ac))
                 (exists (ip)
                    (and (buml:input ac ip)
                         (exists (v f)
                            (and (form:holdsA f xsub)
                                 (form:property-value xa ip v f)))))))))
```

The action-pin-trigger relation links actions (the ac variable) under executions of their containing activities (xa), with other executions (xsub) under the activity execution before which the action input pins are not satisfied and after which they are.

```
(forall (ac xsub xa)
   (iff (form:action-pin-trigger ac xsub xa)
        (and (form:subactivity_occurrence-neq xsub xa)
             (forall (xlsub alsub)
                (if (and (psl:leaf_occ xlsub xsub)
                         (psl:occurrence_of xlsub alsub))
                    (exists (xbsub)
                       (and (= xlsub (psl:successor xbsub alsub))
                            (not (form:action-input-pins-satisfied ac xa xbsub))
                            (form:action-input-pins-satisfied ac xa xlsub)))))))))
```

The take-input relation ensures that executions (the xac variable) remove values from input pins that are values just after another execution is complete (xsub). The other execution brings about pin satisfaction, see action-pin-trigger above. It assumes that the input pin multiplicity upper bound is one or unlimited.

```
(forall (xac ac xa xsub)
  (iff (form:take-input xac ac xa xsub)
       (forall (srootxac arootxac ip ipmax)
          (if (and (psl:root_occ srootxac xac)
                   (psl:occurrence_of srootxac arootxac)
                   (buml:input ac ip)
                   (buml:upper ip ipmax))
              (or (and (= ipmax form:1)
                       (forall (fxsubh v)
                          (if  (and (form:holdsA fxsubh xsub)
                                    (form:property-value xa ip v fxsubh))
                               (form:remove-property-value xa ip v arootxac))))
                  (and (= ipmax buml:*)
                       (forall (fxsubh v)
                          (if (and (form:holdsA fxsubh xsub)
                                   (form:property-value xa ip v fxsubh))
                              (form:remove-property-value xa ip v
                                                            arootxac)))))))))
(forall (n)
    (iff (form:no-incoming-edge-or-one-from-initial n)
         (and (forall (e1 e2)
                (if (and (buml:incoming e1 n)
                         (buml:incoming e2 n))
                    ( = e1 e2)))
              (forall (n2)
                (if (form:flow-trans-fork-merge n2 n)
                    (buml:InitialNode n2))))))

(forall (ac a ip)
   (if (and (buml:Action ac)
            (form:no-incoming-edge-or-one-from-initial ac)
            (buml:input ac ip)
            (form:activity ac a))
       (forall (xa f xsub)
           (if (and (form:classifies a xa f)
                    (form:action-pin-trigger ac xsub xa))
               (exists (xac)
                  (and (form:property-value xa ac xac f)
                       (psl:min_precedes xsub xac a)
                       (form:take-input xac ac xa xsub)))))))
```

10.4.8.4 Action with pins, one incoming control flow from action, optional fork/merge

This subclause applies to the syntactic pattern of an action with pins, and one incoming control flow from another action with any intervening and chained fork and merge nodes (for example, see the Method Call pattern in A.5.11). It requires the target action to execute after the source action when the input pins are provided enough values to meet their lower multiplicity.

The joinable-control-input relation links actions and their executions (the ac0 and xac0 variables respectively) under executions of their containing activities (xa and a, respectively), with other executions (xsub) under the activity execution that satisfy the pins of a target action (ac). The action executions (xac0) and the other pin-satisfying executions (xsub) are paired one-to-one in time order. The pairing begins with action executions and pin-satisfying executions that have no other ones before them (the first part of the disjunction), and the pairs after that are in time order (the second part of the disjunction).

```
(forall (xac0 xsub xa a ac0)
   (iff (form:joinable-control-input ac0 xac0 xsub ac a xa)
        (forall (f)
           (or (and (not (exists (xac00)
                              (and (form:property-value xa ac0 xac00 f)
                                   (psl:min_precedes xac00 xac0 a))))
                    (not (exists (xsub0)
                              (and (form:action-pin-trigger ac xsub0 xa)
                                   (psl:min_precedes xsub0 xsub a)))))
               (exists (xac02 xsub2)
                  (and (form:joinable-control-input ac0 xac02 xsub2 ac a xa)
                       (not (exists (xac00)
                                (and (form:property-value xa ac0 xac00 f)
                                     (psl:min_precedes xac02 xac00 a)
                                     (psl:min_precedes xac00 xac0 a))))
                       (not (exists (xsub0)
                                (and (form:action-pin-trigger ac0 xsub0 xa)
                                     (psl:min_precedes xsub2 xsub0 a)
                                     (psl:min_precedes xsub0 xsub a))))))))))
```

The joined-follows relation links PSL occurrences where two of the occurrences (the s1 and s2 variables) happen before the third (s3) under execution of an activity (a). The third occurrence is constrained to follow no more than one pair of the other two occurrences, and each pair of the other two occurrences to be followed by no more than one of the third (compare to the follows relation in 10.4.7.3).

```
(forall (s1 s2 s3 a)
  (if (form:joined-follows s1 s2 s3 a)
      (and (form:min_precedesA s1 s3 a)
           (form:min_precedesA s2 s3 a))))

(forall (s1 s2 s3 s12 s22 s32 a)
   (if (and (form:joined-follows s1 s2 s3 a)
            (form:joined-follows s12 s22 s32 a))
       (iff (and (= s1 s12)
                 (= s2 s22))
            (= s3 s32))))
```

The joined-action-execution-exists relation establishes existence of executions of actions (the ac variable) under activity executions (xa, an execution of activity a) where an action execution happens after its input pins are satisfied, and after another action execution completes (xac0, an execution of action ac0). An earlier constraint requires the source of control flow (xac0) to execute before the target action (ac), see 10.4.7.3. The constraint below addresses the remaining cases, one where the input pins are satisfied at the time the source of control flow is completed (the first part of the disjunction), and another where the input pins are satisfied sometime after the source of control flow is completed (the second part of the disjunction), see action-pin-trigger and joinable-control-input.

```
(forall (ac ac0 xac0 a xa)
  (iff (form:joined-action-execution-exists ac ac0 xac0 a xa)
       (forall (f)
          (or (and (form:action-input-pins-satisfied ac xa xac0)
                   (exists (xac)
                      (and (form:property-value xa ac xac f)
                           (form:take-input xac ac xa xac0))))
              (and (not (form:action-input-pins-satisfied ac xa xac0))
                   (forall (xsub)
                     (if (and (form:action-pin-trigger ac xsub xa)
                              (form:joinable-control-input ac0 xac0 xsub ac a xa))
                              (exists (xac)
                                 (and (form:property-value xa ac xac f)
                                      (form:follows xac0 xac a)
                                      (form:joined-follows xac0 xsub xac a)
                                      (form:take-input xac ac xa xsub))))))))))
(forall (ac0 ac a)
   (if (and (buml:Action ac0)
            (buml:Action ac)
            (form:max-one-incoming-edge ac)
            (form:flow-trans-fork-merge ac0 ac)
            (form:activity ac a))
       (forall (xa f xac0)
           (if (and (form:classifies a xa f)
                    (form:property-value xa ac0 xac0 f))
               (form:joined-action-execution-exists ac ac0 xac0 a xa)))))
```

10.4.8.5 Action with pins, one incoming control flow from action, decision with decision flow from same action, optional fork/merge

This subclause applies to the syntactic pattern an action with pins, and one incoming control flow from another action with any intervening and chained fork and merge nodes, and one intervening decision node in both the object flows and control flows (for example, see the Do-While pattern in A.4.10). The decision inputs come from output pins on the same action as the source of the control flow. It requires the target action to execute after the source action when the input pins are provided enough values to meet their lower multiplicity.

The guarded-joined-action-execution-exists relation augments the joined-action-execution-exists relation for a syntactic pattern that has a decision node with a decision input from an output pin (the dip variable) and guard value (g). It assumes the output pin has exactly one value, and the guard is a literal value specification. It ensures the value of the output pin is removed.

```
(forall (ac ac0 dip g a)
  (iff (form:guarded-joined-action-execution-exists ac ac0 dip g a)
       (forall (xa f xac0 vdip fxac0)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac0 xac0 f)
                   (form:holdsA fxac0 xac0)
                   (form:property-value xa dip vdip fxac0)
                   (= vdip g))
              (and (form:joined-action-execution-exists ac ac0 xac0 a xa)
                   (exists (amove occmove)
                      (and (form:remove-property-value xa dip vdip amove)
                           (psl:occurrence_of occmove amove)
                           (psl:next_subocc xac0 occmove a))))))))
(forall (ac0 ac dip g a)
   (if (and (buml:Action ac0)
            (buml:Action ac)
            (form:max-one-incoming-edge ac)
            (form:flow-trans-fork-merge-decision ac0 ac dip g)
            (buml:output ac0 dip)
            (form:activity ac a))
       (form:guarded-joined-action-execution-exists ac ac0 dip g a)))
```

10.4.8.6 Action with pins, one incoming control flow from initial, decision with decision flow from initial action in same, optional fork/merge

This subclause applies to the syntactic pattern an action with pins, and one incoming control flow from and initial node with any intervening and chained fork and merge nodes, and one intervening decision node the control flow (for example, see the If Statement pattern in A.4.9). The decision inputs come from output pins on the same action as the source of the control flow. It requires the target action to execute for every execution of the activity when the input pins are provided enough values to meet their lower multiplicity.

```
(forall (i ac dip g ac0 a)
   (if (and (buml:InitialNode i)
            (buml:Action ac)
            (form:max-one-incoming-edge ac)
            (form:flow-trans-fork-merge-decision i ac dip g)
            (buml:output ac0 dip)
            (form:executable-without-input ac0)
            (form:same-syntactic-container i ac)
            (form:same-syntactic-container i ac0)
            (form:activity ac a))
        (form:guarded-joined-action-execution-exists  ac ac0 dip g a)))
```

10.4.9 Invocation Actions

10.4.9.1 Syntax

This subclause specifies additional syntax for invocation actions. The called relation links call actions to behaviors and operations that are called.

```
(forall (ac pd)
   (if (form:called ac pd)
       (and (buml:CallAction ac)
            (form: ProcessDefinition pd))))

(forall (ac po1 po2)
   (if (and (form:called ac po1)
            (form:called ac po2))
       (= po1 po2)))

(forall (ac)
   (if (buml:CallAction ac)
       (exists (po)
          (form:called ac po))))
```

The pin-parameter-match relation links pins and called parameters as derived from pin and parameter ordering in the model.

```
(forall (pn p)
   (if (form:pin-parameter-match pn p)
       (and (buml:Pin pn)
            (buml:Parameter p))))

(forall (pn1 p1 pn2 p2)
   (if (and (form:pin-parameter-match pn1 p1)
            (form:pin-parameter-match pn2 p2))
       (iff (= pn1 pn2)
            (= p1 p2))))
```

```
(forall (pn p)
   (if (form:pin-parameter-match pn p)
       (and (if (form:InputParameter p)
                (buml:InputPin pn))
            (if (form:OutputParameter p)
                (buml:OutputPin pn)))))

(forall (pn p)
   (if (form:pin-parameter-match pn p)
       (forall (m)
          (and (iff (buml:lower pn m)
                    (buml:lower p m))
               (iff (buml:upper pn m)
                    (buml:upper p m))
               (iff (buml:type pn m)
                    (buml:type p m))))))

(forall (pn p)
   (if (form:pin-parameter-match pn p)
       (forall (t)
          (iff (buml:type pn t)
               (buml:type p t)))))

(forall (ac po)
   (if (and (buml:CallAction ac)
            (form:called ac po))
       (forall (pn p)
          (iff (and (form:put ac pn)
                    (form:pin-parameter-match pn p))
               (form:ownedParameter po p)))))

(forall (ac b)
   (if (buml:behavior ac b)
       (buml:type ac b)))

(forall (ac b)
   (if (buml:behavior ac b)
       (form:called ac b)))

(forall (ac op)
   (if (buml:operation ac op)
       (form:called ac op)))

(forall (ac op)
   (if (buml:operation ac op)
       (buml:type ac op)))
```

The pin-property-match relation links pins and properties as derived from pin and signal property ordering in the model.

```
(forall (pn p)
   (if (form:pin-property-match pn p)
       (and (buml:Pin pn)
            (buml:Property p))))

(forall (pn1 p1 pn2 p2)
   (if (and (form:pin-property-match pn1 p1)
            (form:pin-property-match pn2 p2))
       (iff (= pn1 pn2)
            (= p1 p2))))
(forall (pn p)
   (if (form:pin-property-match pn p)
       (forall (m)
          (and (iff (buml:lower pn m)
                    (buml:lower p m))
               (iff (buml:upper pn m)
                    (buml:upper p m))
               (iff (buml:type pn m)
                    (buml:type p m))))))

(forall (ac sig)
   (if (and (buml:SendSignalAction ac)
            (form:signal ac sig))
       (forall (pn p)
          (iff (and (form:argument ac pn)
                    (form:pin-property-match pn p))
               (buml:ownedAttribute sig p)))))
```

10.4.9.2 Semantics

This subclause gives necessary conditions on executions of invocation actions as used in the execution engine (CallBehaviorAction, CallOperationAction, and SendSignalAction).

The change-only-pin relation links actions and their executions under the activity executions, where the action executions only affect spins of the action.

```
(forall (ac xac xa)
    (iff (form:change-only-pin ac xac xa)
         (forall (xsub o p v)
            (if (and (form:subactivity_occurrence-neq xsub xac)
                     (form:achieves-property-value o p v xsub))
                (and (= o xa)
                     (or (buml:input ac p)
                         (buml:input ac p)))))))
```

This ensures activities invoked with call behavior actions transfer values between pins and parameter nodes, and that called function behaviors only affect pins of their calling actions.

```
(forall (ac a b)
    (if (and (buml:CallBehaviorAction ac)
             (form:activity ac a)
             (buml:type ac b))
        (forall (xa xac f)
           (if (and (form:classifies a xa f)
                    (form:property-value xa ac xac f))
               (and (if (buml:Activity b)
                        (form:fill-empty-parameter-node a xa ac xac xac))
                    (if (buml:FunctionBehavior b)
                        (form:change-only-pin ac xac xa)))))))
```

The dispatch relation links objects to operations and behaviors in a PSL state. It is used to determine more detailed behavior (method) based on the thing on which an operation is invoked. It assumes multiple generalization does not affect the choice of method.

```
 (forall (o op b f)
   (if (form:dispatch o op b f)
       (and (buml:Operation op)
            (buml:Behavior b)
            (psl:state f))))

(forall (o op b f)
   (if (form:dispatch o op b f)
       (exists (c)
          (and (form:classifies c o f)
               (form:method op b c)
               (not (exists (c2 b2)
                        (and (not (= c c2))
                             (buml:general c2 c)
                             (form:method op b2 c)
                             (form:classifies c2 o f))))))))
```

The execution-performer relation links executions to things performing those executions.

```
(forall (x o)
  (if (form:execution-performer x o)
      (form:execution x)))

(forall (x o1 o2)
  (if (and (form:execution-performer x o1)
           (form:execution-performer x o2))
      (= o1 o2)))
```

This ensures activities invoked with call operation actions transfer values between pins and parameter nodes, and that called function behaviors only affect pins of their calling actions.

```
(forall (ac a op b tip)
   (if (and (buml:CallOperationAction ac)
            (form:activity ac a)
            (buml:operation ac op)
            (buml:target ac tip))
       (forall (xa xac f fxac to)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa tip to fxac)
                   (form:dispatch to op b fxac))
              (and (form:classifies b xac f)
                   (form:execution-performer xac to)
                   (form:fill-empty-parameter-node a xa ac xac xac))))))
```

The new-object relation links things to PSL occurrences before which the object does not exist.

```
(forall (o occ)
   (if (form:new-object o occ)
       (psl:occurrence occ)))

(forall (o occ)
    (if (form:new-object o occ)
        (and (forall (locc occ2)
                (if (and (psl:leaf_occ locc occ)
                         (psl:earlierEq occ2 locc))
                    (not (exists (f p s o2 c)
                           (and (psl:prior f occ2)
                                (or (form:property-value-sequence o p s f)
                                    (form:property-value-sequence o2 p o f)
                                    (form:classifies c o f)))))))
             (exists (f p s o2 c)
                (and (psl:holdsA f occ)
                     (or (form:property-value-sequence o p s f)
                         (form:property-value-sequence o2 p o f)
                         (form:classifies c o f)))))))
```

The fill-signal-property relation links signal objects to input pins that have values for properties of the signal. It assumes input pin multiplicity upper is one or unlimited.

```
(forall (osig ip ipmax xa xac)
  (iff (form:fill-signal-property osig ip ipmax xa xac)
       (forall (srootxac arootxac p fxac)
          (if (and (psl:root_occ srootxac xac)
                   (psl:occurrence_of srootxac arootxac)
                   (form:pin-property-match ip p)
                   (form:priorA fxac xac))
              (or (and (= ipmax buml:*)
                       (forall (s)
                          (if (form:property-value-sequence xa ip s fxac)
                             (form:set-property-value-sequence osig p s
                                                               arootxac))))
                  (and (= ipmax form:1)
                       (forall (v)
                          (if (form:property-value xa ip v fxac)
                              (form:add-property-value osig p v arootxac))))))))
```

The event-pool relation links things to collections of events sent to them and waiting to be processed.

```
(forall (o osig f)
  (if (form:event-pool o osig f)
      (psl:state f)))
```

This ensures send signal action executions transfer values between pins and signal properties, and that the signal is in the event pool of the target.

```
(forall (ac a sig tip)
    (if (and (buml:SendSignalAction ac)
             (form:activity ac a)
             (buml:signal ac sig)
             (buml:target ac tip))
        (forall (xa xac f fxac to)
           (if (and (form:classifies a xa f)
                    (form:property-value xa ac xac f)
                    (form:priorA fxac xac)
                    (form:property-value xa tip to fxac))

             (exists (fxac2 osig)
                  (and (form:holdsA fxac2 xac)
                        (form:classifies sig osig fxac2)
                        (form:new-object osig xac)
                        (forall (ip ipmax)
                            (if (and (buml:argument ac ip)
                                     (buml:upper ip ipmax))
                                (form:fill-signal-property osig ip ipmax xa xac)))
                        (form:event-pool to osig fxac2)))))))
```

10.4.10 Object Actions (Intermediate)

This subclause specifies necessary conditions on executions of intermediate object actions as used in the execution engine (CreateObjectAction, TestIdentityAction, ReadSelfAction, and ValueSpecificationAction).

```
(forall (ac a c op)
   (if (and (buml:CreateObjectAction ac)
            (form:activity ac a)
            (buml:classifier ac c)
            (buml:result ac op))
       (forall (xa xac f)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f))
              (exists (fxac o)
                 (and (form:holdsA fxac xac)
                      (form:classifies c o fxac)
                      (form:new-object o xac)
                      (form:property-value xa op o fxac)))))))
```

This covers testing equivalence of datatype values, which the UML TestIdentityAction does not.

```
(forall (ac a ip1 ip2 op)
   (if (and (buml:TestIdentityAction ac)
            (form:activity ac a)
            (buml:first ac ip1)
            (buml:first ac ip2)
            (buml:result ac op))
       (forall (xa xac f v1 v2 fxac)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa ip1 v1 fxac)
                   (form:property-value xa ip2 v2 fxac))
              (exists (fxac2)
                 (and (form:holdsA fxac2 xac)
                      (if (or (= v1 v2)
                              (form:same-string v1 v2))
                          (form:property-value xa op form:true fxac2))
                      (if (not (= v1 v2))
                          (form:property-value xa op form:false fxac2))))))))
(forall (ac a op)
   (if (and (buml:ReadSelfAction ac)
            (form:activity ac a)
            (buml:result ac op))
       (forall (xa xac f xsuper o)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
```

```
                  (form:subactivity_occurrence-neq xac xsuper)
                  (form:execution-performer o xsuper)
                  (not (exists (xsuper2 o2)
                           (and (form:subactivity_occurrence-neq xac xsuper2)
                                (form:subactivity_occurrence-neq xsuper2 xsuper)
                                (form:execution-performer o2 xsuper2)))))
              (exists (fxac)
                 (and (form:holdsA fxac xac)
                      (form:property-value xa op o fxac)))))))
(forall (ac a vs op)
   (if (and (buml:ValueSpecificationAction ac)
            (form:activity ac a)
            (buml:value ac vs)
            (buml:result ac op))
       (forall (xa xac f)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f))
              (exists (fxac)
                 (and (form:holdsA fxac xac)
                      (forall (v)
                         (if (buml:value vs v)
                             (form:property-value xa op v fxac)))
                      (if (buml:LiteralNull vs)
                          (form:property-value xa op form:null fxac))
                      (if (buml:InstanceValue vs)
                          (forall (i)
                             (if (buml:instance vs i)
                                 (form:property-value xa op i fxac))))))))))
```

10.4.11 Structural Feature Actions

This subclause specifies necessary conditions on executions of intermediate object actions as used in the execution model (ReadStructuralFeatureAction, ClearStructuralFeatureAction, AddStructuralFeatureAction, RemoveStructuralFeatureAction).

```
(forall (ac a p oip op)
   (if (and (buml:ReadStructuralFeatureAction ac)
            (form:activity ac a)
            (buml:structuralFeature ac p)
            (buml:object ac oip)
            (buml:result ac op))
       (forall (xa xac f o fxac v)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa oip o fxac)
                   (form:property-value o p v fxac))
              (exists (fxac2)
                 (and (form:holdsA fxac2 xac)
                      (form:property-value xa op v fxac2)))))))
(forall (ac a p oip op)
   (if (and (buml:ClearStructuralFeatureAction ac)
            (form:activity ac a)
            (buml:structuralFeature ac p)
            (buml:object ac oip)
            (buml:result ac op))
       (forall (xa xac f o fxac)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa oip o fxac))
              (and (not (exists (fxac2 v)
                          (and (form:holdsA fxac2 xac)
                               (form:property-value o p v fxac2))))
                   (exists (fxac2)
                      (and (form:holdsA fxac2 xac)
                           (form:property-value xa op o fxac2))))))))

(forall (ac a p oip vip)
    (if (and (buml:AddStructuralFeatureAction ac)
             (form:activity ac a)
             (buml:structuralFeature ac p)
             (buml:object ac oip)
             (buml:value ac vip))
       (forall (xa xac f o fxac v)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa oip o fxac)
                   (form:property-value xa vip v fxac))
              (and (form:property-value o p v fxac)
                   (forall (iraip irav)
                     (if (and (buml:isReplaceAll ac iraip)
                              (form:property-value xa iraip irav fxac)
                              (= iraip form:true))
                         (not (exists (fxac2 v2)
                                (and (form:holdsA fxac2 xac)
                                     (form:property-value o p v2 fxac2)
                                     (not (= v v2)))))))
                   (forall (iaip iav s sl sl1 n)
                     (if (and (buml:insertAt ac iaip)
                              (form:property-value xa iaip iav fxac)
                              (form:property-value-sequence o p s fxac)
                              (form:sequence-length s sl)
                              (form:add-one sl sl1)
                              (if (= iav buml:*)
                                  (= n sl1))
                              (if (not (= iav buml:*))
                                  (= n iav)))
                         (exists (fxac2 s2)
                           (and (or (= iav buml:*)
                                    (and (form:WholeNumber iav)
                                         (form:less-than iav sl1)))
                                (form:holdsA fxac2 xac)
                                (form:property-value-sequence o p s2 fxac2)
                                (form:in-position-count s2 v n)
                                (forall (nless v2)
                                   (if (and (form:WholeNumber nless)
                                            (form:less-than nless n)
```

```
                                                      (form:in-position-count s v2 nless))
                                                 (form:in-position-count s2 v2 nless)))
                                          (if (form:WholeNumber iav)
                                              (forall (nmore v2 nmore1)
                                                (if (and (form:WholeNumber nmore)
                                                         (form:less-than n nmore)
                                                          (form:less-than nmore sl1)
                                                          (form:in-position-count s v2 nmore)
                                                          (form:add-one nmore nmore1))
                                                    (form:in-position-count s2 v2
                                                                          nmore1))))))))))))
```

This assumes isRemoveDuplicates on the action is false.

```
(forall (ac a p oip vip)
    (if (and (buml:RemoveStructuralFeatureAction ac)
             (form:activity ac a)
             (buml:structuralFeature ac p)
             (buml:object ac oip)
             (buml:value ac vip))
        (forall (xa xac f o fxac v)
           (if (and (form:classifies a xa f)
                    (form:property-value xa ac xac f)
                    (form:priorA fxac xac)
                    (form:property-value xa oip o fxac)
                    (form:property-value xa vip v fxac))
               (and (if (not (exists (raip)
                                (buml:insertAt ac raip)))
                        (not (exists (fxac2)
                                (and (form:holdsA fxac2 xac)
                                     (form:property-value o p v fxac)))))
                    (forall (raip n s sl sl1)
                       (if (and (buml:removeAt ac raip)
                                (form:property-value xa raip n fxac)
                                (form:property-value-sequence o p s fxac)
                                (form:sequence-length s sl)
                                (form:add-one sl sl1))
                           (exists (fxac2 s2)
                              (and (form:holds fxac2 xac)
                                   (form:property-value-sequence o p s2 fxac2)
                                   (form:WholeNumber n)
                                   (form:less-than n sl1)
                                   (forall (nless v2)
                                      (if (and (form:WholeNumber nless)
                                               (form:less-than nless n)
                                               (form:in-position-count s v2 nless))
                                          (form:in-position-count s2 v2 nless)))
                                   (forall (nmore v2 nmore1)
                                     (if (and (form:WholeNumber nmore)
                                              (form:less-than n nmore)
                                              (form:less-than nmore sl1)
                                              (form:in-position-count s v2 nmore)
                                              (form:add-one nmore1 nmore))
                                         (form:in-position-count s2 v2
                                                                nmore1)))))))))))
```

10.4.12 Object Actions (Complete)

This subclause specifies necessary conditions on executions of complete object actions as used in the execution model (ReadIsClassifiedObjectAction and StartObjectBehaviorAction).

This assumes the isDirect attribute on the action is false.

```
(forall (ac a ip c op)
   (if (and (buml:ReadIsClassifiedObjectAction  ac)
            (form:activity ac a)
            (buml:object ac ip)
            (buml:classifier ac c)
            (buml:result ac op))
       (forall (xa xac f o fxac)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa ip o fxac))
              (exists (fxac2)
                 (and (form:holdsA fxac2 xac)
                      (if (form:classifies c o fxac)
                          (form:property-value xa op form:true fxac2))
                      (if (not (form:classifies c o fxac))
                          (form:property-value xa op form:false fxac2))))))))
```

This assumes the object input is not a behavior, that no arguments are passed, and the action is not synchronous.

```
(forall (ac a ip)
   (if (and (buml:StartObjectBehaviorAction  ac)
            (form:activity ac a)
            (buml:object ac ip))
       (forall (xa xac f o fxac c b)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:priorA fxac xac)
                   (form:property-value xa ip o fxac)
                   (form:classifies c o fxac)
                   (buml:classifierBehavior c b))
              (exists (xb srootxb)
                 (and (form:classifies b xb f)
                      (form:execution-performer xb o)
                      (psl:root_occ srootxb xb)
                      (form:subactivity_occurrence-neq srootxb xac)))))))
```

10.4.13 Accept Event Action

This subclause specifies necessary conditions on executions of AcceptEventAction. The getNextEvent relation links things and triggers with signal objects in the thing's event pool in a PSL state.

```
(forall (o osig tr f)
   (if (form:getNextEvent o osig tr f)
       (exists (ev sig)
          (and (form:event-pool o osig f)
               (buml:event tr ev)
               (buml:signal ev sig)
```

```
                (form:classifies sig osig f)))))
(forall (o osig1 tr osig2 f)
   (if (and (form:getNextEvent o osig1 tr f)
            (form:getNextEvent o osig2 tr f))
       (= osig1 osig2)))

(forall (ac)
   (if (buml:AcceptEventAction ac)
       (buml:type ac form:AcceptEventBehavior)))

(forall (ac a tr op)
   (if (and (buml:AcceptEventAction ac)
            (form:activity ac a)
            (buml:trigger ac tr)
            (buml:result ac op))
       (forall (xa xac f o xlac fxlac)
          (if (and (form:classifies a xa f)
                   (form:property-value xa ac xac f)
                   (form:execution-performer xac o)
                   (psl:leaf_occ xlac xac)
                   (form:priorA fxlac xlac))
              (exists (osig)
               (and (form:getNextEvent o osig tr fxlac)
                    (exists (fxac2)
                      (and (psl:holds fxac2 xac)
                           (not (form:event-pool o osig fxac2))
                           (form:property-value xa op osig fxac2)))
                     (not (exists (xac2 lxac2)
                             (and (form:classifies form:AcceptEventBehavior xac2 f)
                                  (form:execution-performer xac2 o)
                                 (not (= xac2 xac))
                                  (psl:leaf_occ lxac2 xac2)
                                  (psl:leaf_occ lxac2 xac))))))))))
```

This page intentionally left blank

Annex A Java to UML Activity Mapping
(normative)

A.1 General

The specifications for the methods of operations in the execution model in Clause 8 are written as Java code. However, as discussed in 8.1, this Java code is to be interpreted as a surface syntax for UML activity model. This annex defines the normative mapping from the Java syntax used in the execution model to UML activity models.

Subclause A.2 defines the correspondence between type names in the Java code and types in UML. The remaining subclauses map Java behavioral code to UML activity models. In each case, the mapping is giving in terms of a pattern of Java code and the pattern for the corresponding UML activity model (except for the case of a Java while loop, which is mapped to an equivalent Java do-while loop and, from that, to a UML activity model). The rules for the mapping are also described textually. The textual rules are intended to be used in conjunction with the graphical depiction of the mapping.

This mapping does not cover the entire Java language. Rather, there are specific conventions, noted in the following subclauses, which must be followed in the Java code in order to allow it to be mapped to UML. Further, the result of this mapping is only subset of the full set of possible UML activity models. This subset defines the behavioral modeling capabilities included in the *Base UML* (or "bUML") subset of fUML that is used to write the fUML execution model. Clause 10 gives the base semantics for the bUML subset.

A.2 Type Names

Table A.1 defines the mapping from type names mentioned in the Java code to corresponding UML types.

Note that Java variables typed by a class are always allowed to have the empty value "null." This is considered to correspond, in UML, to the empty case of no values. Thus, all Java types are mapped to UML multiplicity elements with a lower bound of 0.

Further, types with names of the form "…List" map to UML multiplicity elements with an unlimited upper bound. See A.6 for more on list types.

Table A.1 - Java to UML Type Name Mapping

Java	UML
Primitive types	
int	Integer
float	Real
boolean	Boolean
String	String
fUML.Syntax.UnlimitedNatural	UnlimitedNatural
Classes	
<class name>	<class name> [0..1]

Table A.1 - Java to UML Type Name Mapping

Java	UML
<package name>.<class name>	<package name>::<class name> [0..1] (**Note:** "." separators are replaced by "::" in the package name)
Lists	
<type name>List	<type name> [*] {ordered, non-unique}

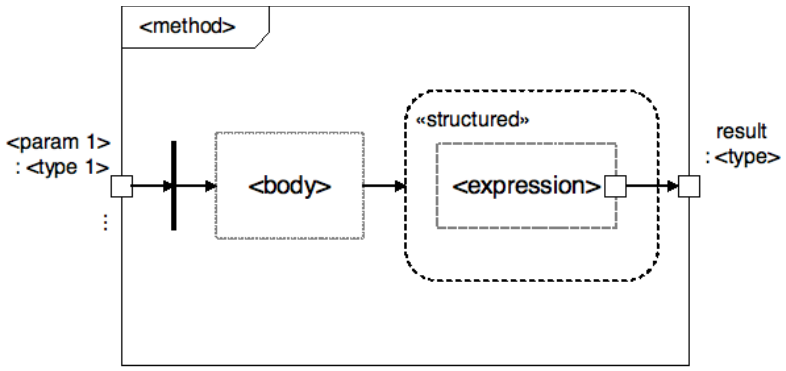
A.3 Method Declaration

Java

```
public <type> <method>
   (<type 1> <param 1>, ...)
{
    <body>
    return <expression>;
}
```

- A method with a non-void type must have a single return statement at the end of its body. A void method may not have any return statements, except that a method with no other statements may have a single "return" statement in its body.

UML



- A method maps to an activity with the corresponding operation as its specification.
- The parameters of the method map to input parameters of the activity, with corresponding activity parameter nodes. The result type of the method, of other than void, maps to a single result parameter of the activity, with a corresponding activity parameter node. If the method has a void type, the activity has no result parameter.
- The body of the method is mapped as a sequence of statements (see A.4.1).
- Each input activity parameter node is connected by an object flow to a fork node. A use of a method parameter in the body maps to an object flow from the fork node connected to the corresponding input activity parameter node into the mapping of the body.

- A return statement (with an expression) maps to a structured activity node containing the mapping of the return expression (see A.5), with a control flow dependency on the mapping of the final statement of the body (unless this is empty). Object flows may flow from within the mapping of the body into the mapping of the expression. An object flow connects the result pin of the expression to the result activity parameter node. (A return statement for an otherwise empty method is not mapped to anything.)

A.4 Statements

The following mappings are for statements and sequences of statements that appear in the bodies of methods and structured statements. Statements often have embedded expressions, which are mapped according to the mappings given in A.5.

A.4.1 Statement Sequence

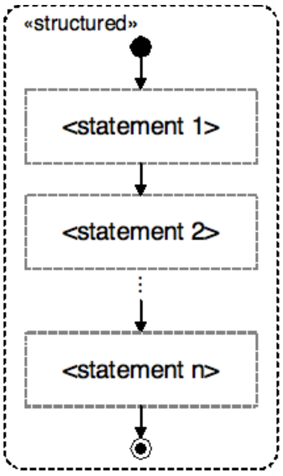
Java

```
<statement 1>;
<statement 2>;
...

<statement n>
```

- Allowable statements include only those with a form that has a mapping defined in the remainder of this subclause.

UML



- The mapping of a sequence of statements consists of a structured activity node that contains the mapping of each statement (as given in the remainder of this subclause).
- The mapping of the first statement in the sequence has an incoming control flow from an initial node. The mapping of each subsequent statement has a control flow from the mapping of the previous statement. The mapping of the last statement has an outgoing control flow to an activity final node. (An empty sequence maps to a structured activity node with an initial node connected directly to a final node.)
- Object flows from within the mapping of one statement may flow into the mapping of a subsequent statement.

Notes

- The actual sources and targets of the control flows within the statement mappings are noted in the mapping for each kind of statement.

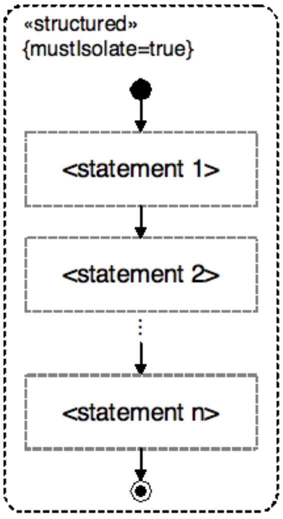
A.4.2 Statement Sequence (isolated)

Java

```
_beginIsolation();
    <statement 1>;
    <statement 2>;
    …
    <statement n>;
_endIsolation();
```

- A set of statements the must run in "isolation" are represented by a sequence of statements, the first statement of which is a call to the "_beginIsolation()" method and the last statement is a call to the "_endIsolation()" method.
- A user class may not define methods called "_beginIsolation" or "_endIsolation."

UML



- The sequence of statements in the block is mapped in exactly the same way as for a normal sequence of statements (see A.4.1), but the enclosing structured activity node has mustIsolate=true.

A.4.3 Local Variable Declaration

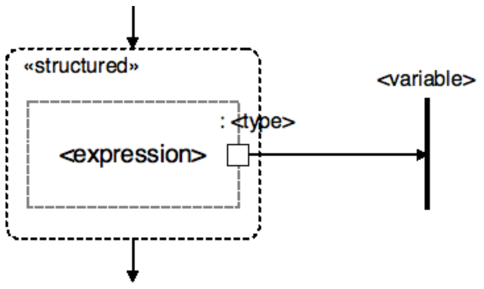
Java

```
<type> <variable> = <expression>;
```

- A local variable declaration is required to have an initialization expression.

- It is not permitted to reassign the value of a local variable, except as specifically allowed in the context of an if statement or loop (see A.4.9 and A.4.10).

UML



- A local variable declaration maps to fork node that receives an object flow from the result of the mapping of the initialization expression (see A.5).
- The mapping of the initialization expression is nested inside a structured activity node. Incoming and outgoing control flows (if any) attach to the structured activity node.

Notes

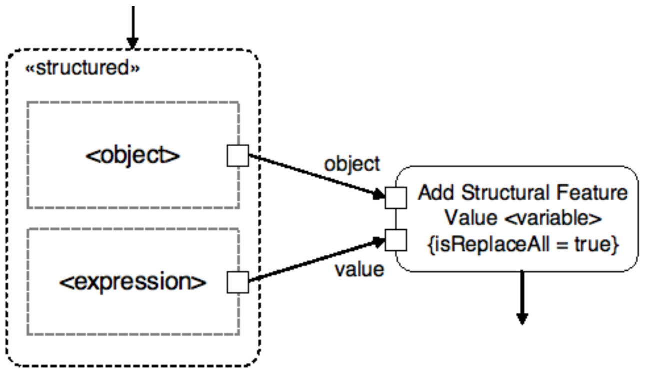
- The use of the fork node models the ability to read the value of a local variable multiple times.
- If the local variable is re-assigned as part of a subsequent if statement or loop, then uses of the variable after that point will be mapped to flows from a different fork node than the one resulting from the mapping of the variable declaration (see A.4.1 and A.4.2).

A.4.4 Instance Variable Assignment (non-list)

Java

```
<object>.<variable> = <expression>;
```

UML


«structured»
<object>
<expression>
object
value
Add Structural Feature Value <variable> {isReplaceAll = true}

- The assignment of a non-list instance variable maps to an add structural feature value action with isReplaceAll = true. (For instance variables of a list type, see A.4.5).

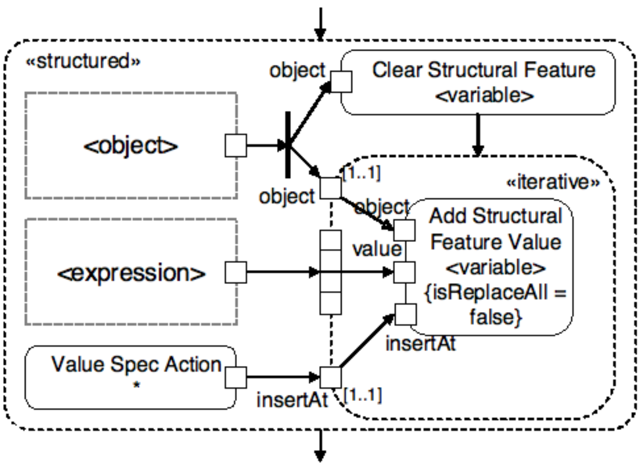
- The object and assigned expressions map as given in A.5. Their mappings are nested in a structured activity node.
- The object input pin of the add structural feature value action is connected by an object flow to the result pin of the mapping of the object expression.
- The value pin of the add structure feature value action is connected by an object flow to the result pin of the mapping of the assigned expression.
- An incoming control flow (if any) attaches to the structured activity node containing the expression mappings. An outgoing control flow (if any) attaches to the add structural feature action.

A.4.5 Instance Variable Assignment (list)

Java

```
<object>.<variable> = <expression>;
```

UML



- The assignment of a list instance variable maps to a clear structural feature action followed by an expansion region containing an add structural feature value action with isReplaceAll = false. (For assignment of an instance variable of a non-list type, see A.4.4).
- The object and assigned expressions map as given in A.5.
- The object input pin of the clear structure feature action and an input pin (multiplicity [1..1]) of the expansion region are connected by object flows to a fork node that is connected by an object flow to the result pin of the mapping of the object expression.
- An input expansion node on the expansion region is connected by an object flow to the result pin of the mapping of the assigned expression.
- An input pin on the expansion region is connected by an object flow to the result ping of a value specification action that produces an unlimited natural * (unbounded) value.
- Inside the expansion region, the object input pin of the add structural feature value action is connection by an object flow to the object input pin of the expansion region, its value pin is connected by an object flow to the expansion node and its insertAt pin is connected by an object flow to the insertAt input pin of the expansion region.

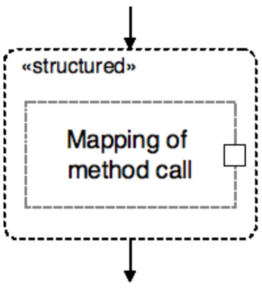
- The expression mappings, clear structural feature action and expansion region are all nested in a structured activity node. Incoming and outgoing control flows (if any) attach to the structured activity node.

A.4.6 Method Call Statement

Java

```
<object>.<method>(<argument 1>,…);
```

UML



- A statement containing only a method call maps as a method call expression (see A.5.11). The result pin of the mapping (if any) has no outgoing object flow.
- The mapping of the method call is nested inside a structured activity node. Incoming and outgoing control flows (if any) attach to the structured activity node.
- A statement containing only a super call maps in a similar manner, but with a super call expression (see A.5.12) rather than a method call expression.

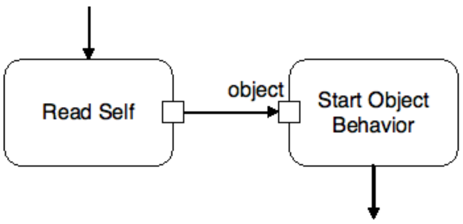
A.4.7 Start Object Behavior

Java

```
_startObjectBehavior();
```

- A class may not define a user operation called "_startObjectBehavior."
- The _startObjecttBehavior method may not be called explicitly on any other object.

UML



- A _startObjectBehavior call maps to a start object behavior action.

- The object input pin of the start object behavior action is connected by an object flow to the result pin of a read self action.
- An incoming control flow (if any) attaches to the read self action. An outgoing control flow (if any) attaches to the start object behavior action.

Notes

- An object can have at most one classifier behavior. The _startObjectBehavior method starts this in a separate thread and returns immediately.
- This mapping is an exception to the normal "Method Call Statement" mapping of A.4.6.

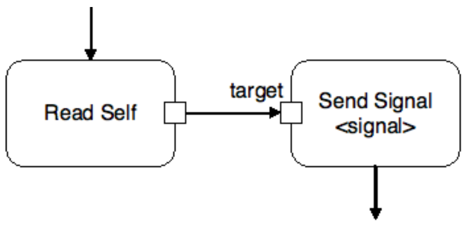
A.4.8 Signal Send

Java

```
_send(new <signal>());
```

- The constructor for a signal may not have any arguments. (Signals with attributes are not allowed.)

UML



- A _send method call maps to a send signal action for the constructed signal.
- The target input pin of the send signal operation is connected by an object flow to the result pin of a read self action.
- An incoming control flow (if any) attaches to the read self action. An outgoing control flow (if any) attaches to the send signal action.

Notes

- This is an exception to the normal "Method Call Statement" mapping of A.4.6.
- The classifier behavior of the class containing the method making the _send call must have an accept event action for the signal.

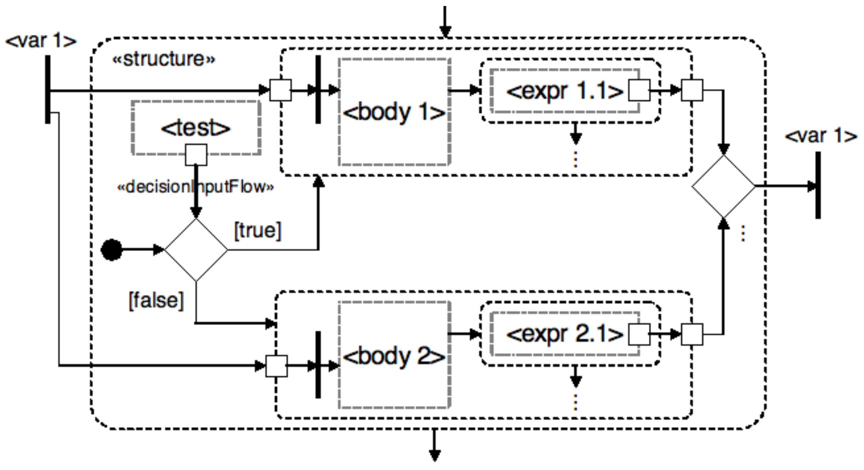
A.4.9 If Statement

Java

```
if (<test>) {
    <body 1>
    <var 1> = <expr 1.1>;
    …
} else {
    <body 2>
    <var 1> = <expr 2.1>;
    …
}
```

- At the end of the body of each branch of the if statement, there may be assignment statements for local variables declared outside the if statement.

UML



- An if statement maps to a decision node with two outgoing control flows, one with the guard “true” and one with the guard “false” and an incoming control flow from an initial node.
- The decision node has a “decision input” data flow from the result pin of the mapping of the test expression (see A.5).
- Each body maps as a structured activity node containing the mapping of a sequence of statements (see A.4.1). The “true” control flow from the decision node connects to the structured activity node for the first body. The “false” control flow similarly connects to the structured activity node for the second (“else”) body.
- The structured activity nodes for each branch have input and output pins corresponding to the variables assigned in either branch. Object flows connect the source for each variable to the corresponding input pins and each input pin to a fork node within the structured activity node for the branch. Object flows connect the two output pins corresponding to a variable (one from each branch) to a merge node, which then has an object flow to a fork node. The fork node is acts as the source for all uses of the variable subsequently to the if statement.

- Each variable assignment maps to a structured activity node containing the mapping of the assigned expression. The first structured activity node has a control flow dependency on the mapping of the last statement of the branch body (if any) and each subsequent node has a control flow dependency on the previous node. The result pin of each expression has an object flow to the output pin for the branch corresponding to the variable being assigned. If a variable is not assigned in a branch, then the input pin for the variable is connect by an object flow directly to the output pin, within the structured activity node for the branch. If a variable is used in a subsequent assignment expression, then a fork node must be inserted to fork the object flow out of the expression result to both the branch output pin and any subsequent variable use(s).
- The input pins of a structured activity node for a branch act as the source for all uses of the corresponding variables within the branch. For any other variable uses, object flows may flow directly into the mappings of the parts of the if statement. Object flows from within the mapping of the body may flow into the mappings of the expressions.
- If the if statement has no else branch, but there are variable assignments in the "true" branch, then there is still a structured activity node for the "false" branch, with all input pins connected to output pins. If there is no else branch and no variable assignments, then the "false" branch structured activity node may be replaced by an activity final node.
- Incoming and outgoing control flows (if any) attach to the outermost structured activity node.

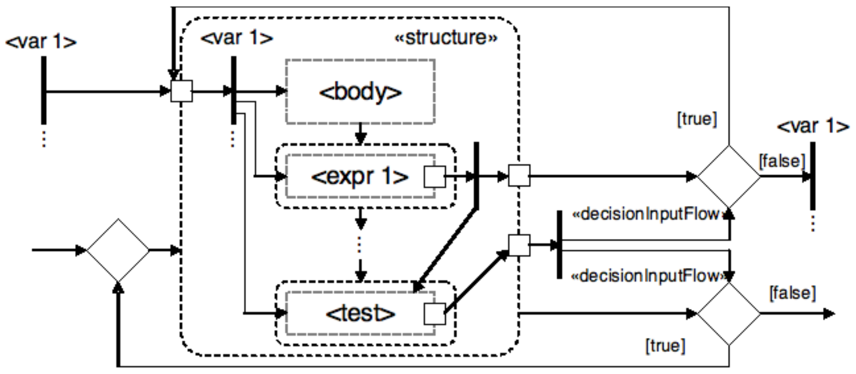
A.4.10 Do-While Loop

Java

```
do {
    <body>
    <var 1> = <expr 1>;
    …
} while (<test>)
```

- A do-while loop may contain variable assignments at the end of its body for local variables declared outside the loop.

UML



- A do-while loop maps to a structured activity node with a looping control structure outside it, as shown above. An incoming control flow comes into the merge node shown on the left, and the outgoing control flow is the "false" flow out of the decision node shown on the bottom right.
- The body of the do-while loop maps as a sequence of statements (see A.4.2).

- Every variable referenced in the body of the while loop (whether it is assigned or not) has corresponding input and output pins on the structured activity node for the loop. The input pin for the variable is connected by an object flow to the mapping for the variable from before the loop. Inside the structured activity node for the loop, each loop variable input pin is connected by an object flow to a fork node. This fork node is used as the source for the mapping of all uses of the variable within the loop (unless, possibly, if the variable is re-assigned within the loop – see below).
- Each variable assignment maps to a structured activity node containing the mapping of the assigned expression. The first structured activity node has a control flow dependency on the mapping of the loop body and each subsequent node has a control flow dependency on the previous node. The result pin of each expression is connected by an object flow to a fork node which then has an object node to the output pin of the outer structured activity node corresponding to the variable being assigned. (If there is no assignment for a variable within the loop, then there is an object flow that connects directly from the fork node for the variable within the loop's structured activity node to the output pin for the variable.)
- The test expression maps to a structured activity node containing the mapping of the expression. There is a control flow from the structured activity node for the mapping of the last variable assignment expression to the structured activity node for the test expression. (If there are no variable assignments, the control flow comes from the mapping of the body.) The result pin of the test expression is connected by an object flow to an output pin of the outer structured activity node for the loop.
- If an assignment or test expression uses a variable previously assigned, then that use maps to an object flow from the fork node attached to the result pin of the assignment expression, rather than the fork node attached to the loop input pin.
- The test result output pin of the structured activity node for the loop is connected to a fork node outside the structured activity node, which is then connected by an object flow to the decision input flow of the loop control decision node. The output pin for each loop variable is connected by an object flow to a decision node. The decision input flow for the decision node is an object flow from the test result fork node. The "true" outgoing flow from the decision node connects back to the corresponding input pin and the false flow connects to a fork node, which is used as the source of the variable for all mappings of expressions after the while loop.

A.4.11 While Loop

Java

```
while (<test>) {
    <body>
    <var 1> = <expr 1>;
    …
}
```

- A while loop may contain variable assignments at the end of its body for local variables declared outside the loop.

Equivalent Java

```
if (<test>) {
    do {
        <body>
        <var 1> = <expr 1>;
        …
    } while (<test>)
}
```

- A while loop maps as if it was coded as a do-while loop (see A.4.10) nested in an if statement (see A.4.9), as shown above.

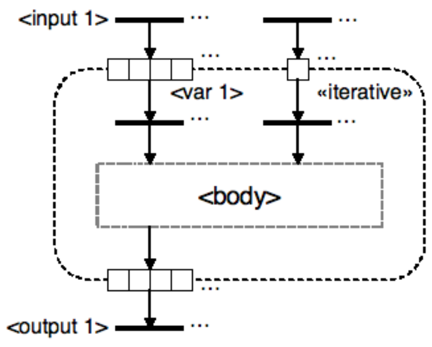
A.4.12 For Loop (iterative)

Java

```
<output type 1> <output 1> = new <output type 1>();
…
for (int <index> = 0;
     <index> < <input 1>.size(); i++) {
  <type 1> <var 1> = <input 1>.getValue(<index>);
  …
  <body>
}
```

- An iterative for loop must have a locally declared index variable of type "int" that is sequentially incremented (see also A.4.13).
- The body of the for loop must begin with one or more loop variable declarations, each initialized by an access to a different list variable with the loop index variable. The list variables must be declared outside the loop and all have list types (see A.6). The loop index variable may not otherwise be used in the body of the loop.
- The for loop must be indexed based on the size of the first list variable, as shown above.
- The for loop must not have any assignment statements at the end of its body.
- The body of the for loop may include nested statements of the form "<output n>.addValue(…)," where "<output n>" is a variable of a list type declared outside the loop and initialized to an empty list of the appropriate type.

UML



- A for loop with the structure given above is mapped to an iterative expansion region.
- The local loop variables map to input expansion nodes on the expansion region. The expansion node for the variable is connected outside the expansion region by an object flow to the mapping for the corresponding list variable from before the loop. It is connected inside the expansion region to a fork node. A reference to a loop variable within the loop body maps to an object flow from the fork node connected to the corresponding expansion node.

- For any variable declared outside the loop and referenced within the body of the loop, other than the local loop variables as defined above, there is a corresponding input pin on the expansion region. The input pin is connected outside the expansion region by an object flow to the fork node corresponding to the variable. The input pin is connected inside the expansion region to a fork node, which is then used as the source for references to the variable within the mapping of the body of the loop.
- The body of the loop maps as a sequence of statements (see A.4.1) nested in the expansion region.
- If there are any "addValue" statements within the body of the loop, then there is an output expansion node for each referenced output list variable. Each "addValue" statement maps to an object flow from the result of the argument expression of the "addValue" call to the appropriate output expansion node. Each output expansion node is connected by an object flow to a fork node that is used as the source for references to the corresponding output list variable in any subsequent statements.

Notes

- The mapping for the element variables is an exception to the normal rules for list indexing (see A.6.7) and for variable use (see A.5.1).

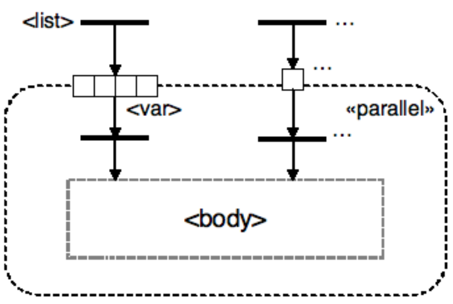
A.4.13 For Loop (parallel)

Java

```
for (Iterator <iter> = <list>.iterator();
     <iter>.hasNext();) {
  <type> <var> = (<type>)(<list>.next());
  …
  <body>
}
```

- A parallel for loop must be indexed by an iterator based on a list variable (see A.5.6). The list variable must be declared outside the loop and have a list type (see A.6).
- The body of the for loop must begin with exactly one variable declaration, initialized by an access to the loop iterator.
- The for loop must not have any assignment statements at the end of its body.
- The behavior of the body must not depend on the specific order in which list items are returned.

UML



- A for loop with the structure given above is mapped to a parallel expansion region.
- The local loop variable maps to a single input expansion node on the expansion region. (There are no output expansion nodes.) The expansion node for the variable is connected outside the expansion region by an object flow to the mapping for the corresponding list variable from before the loop. It is connected inside the expansion region to a fork node. A reference to a loop variable within the loop body maps to an object flow from the fork node connected to the corresponding expansion node.
- For any variable declared outside the loop and referenced within the body of the loop, other than the local loop variables as defined above, there is a corresponding input pin on the expansion region. The input pin is connected outside the expansion region by an object flow to the fork node corresponding to the variable. The input pin is connected inside the expansion region to a fork node, which is then used as the source for references to the variable within the mapping of the body of the loop.
- The body of the loop maps as a sequence of statements (see A.4.1).

Notes

- The Java code will execute the body iterations in a specific sequential order, but the behavior of the Java is not allowed to depend on what that order actually is.
- The mapping for the element variables is an exception to the normal rules for list indexing (see A.6.7) and for variable use (see A.5.6).

A.5 Expressions

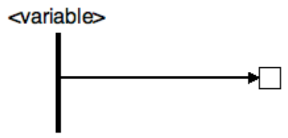
The following mappings are for expressions that are embedded within statements. Each expression maps to a fragment of an activity model that has a distinguished "result pin" (with the exception of the mapping of A.5.1). It is this result pin to which an object flow may be connected to obtain the output of the expression.

A.5.1 Local Variable or Method Parameter Use

Java

```
<variable>
```

UML



The use of a local variable or method parameter in an expression maps to an object flow from the fork node corresponding to the variable or parameter to an input pin of the mapping of the remainder of the enclosing expression.

Notes

- The fork node may result from the mapping of a method parameter, directly from the mapping of the declaration of the variable (see above), from the mapping of the output of an if statement or a loop (see A.4.9 and A.4.10) or from the mapping of the loop variable of a fork node.

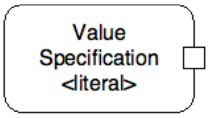
A.5.2 Literal

Java

```
<literal>
```

- The literal must be an integer, a boolean, a string, or an UnlimitedNatural.
- An UnlimitedNatural literal is created using a constructor expression of the form "new fUML.Syntax.UnlimitedNatural(n)," where n is a non-negative integer or -1 (used to represent "*").
- The integer value of an UnlimitedNatural value "x" is obtained by an expression of the form "x.value."

UML



- A literal is mapped to a value specification action with a corresponding literal value. The result output pin of the value specification action becomes the result pin of the mapping.

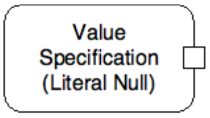
A.5.3 Null

Java

```
null
```

- A null value may not be used for a list type.

UML



- A null value maps to a value specification action for a literal null. The result output pin of the value specification action becomes the result pin of the mapping.

Notes

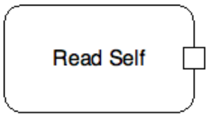
- All class types in Java allow "null" values. Such types map to types with "optional" multiplicity [0..1] in UML (see A.2). Java "null" is used to represent the case of "no value" (0 cardinality) allowed by this multiplicity. A value specification for a literal null places no values on its output pin when it executes, corresponding to the 0 cardinality case.
- Since a Java "null" maps to "no value" in UML, testing for a null value requires a special mapping (see A.5.10).
- For the mapping of an empty list, see A.6.4.

A.5.4 This

Java

```
this
```

UML



- A use of “this” maps to a read self action whose result pin is the result pin for the expression mapping.

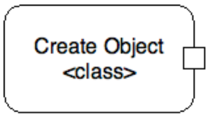
A.5.5 Constructor Call

Java

```
new <class>()
```

- Constructor calls are not allowed to have arguments.

UML



- A constructor call maps to a create object action for the named class. The result output pin of the create object action becomes the result pin for the mapping.

Notes

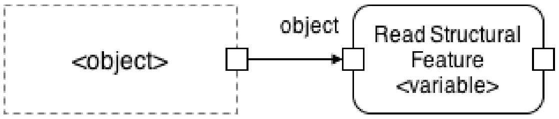
- This mapping does not apply to the case of the class being a list type (see A.6.4 for the construction of an empty list).
- This mapping does not apply to the case of creating an UnlimitedNatural value (see A.5.2).

A.5.6 Instance Variable Use

Java

```
<object>.<variable>
```

UML



- The use of an instance variable within an expression maps to a read structural feature action for the attribute corresponding to the instance variable. The result output pin of the read structural feature action becomes the result pin for the mapping.
- The object input pin of the read structural feature action is connected by an object flow to the result pin of the mapping of the expression evaluating to the target object.

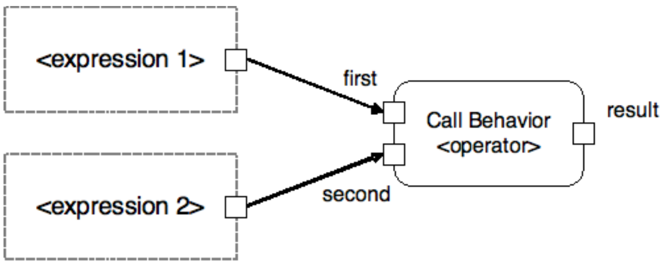
A.5.7 Operator Expression

Java

```
<expression 1> <operator> <expression 2>
```

- The operator must be an integer arithmetic operator or a boolean relational operator other than equals or not equals (for testing equality, see A.5.8 and A.5.9).

UML



- An infix operator expression maps to a call behavior action for the primitive behavior corresponding to the operator (chosen from the Foundational Model Library, see 9.3). The result output pin of the call behavior action becomes the result pin of the mapping.
- The first argument input pin of the call behavior action is connected by an object flow to the result pin of the mapping of the left sub-expression. The second argument input pin of the call behavior action is connected by an object flow to the result pin of the mapping of the right sub-expression.
- A prefix operator is mapped similarly, except that there is only one sub-expression and only one argument input pin to the call behavior action.

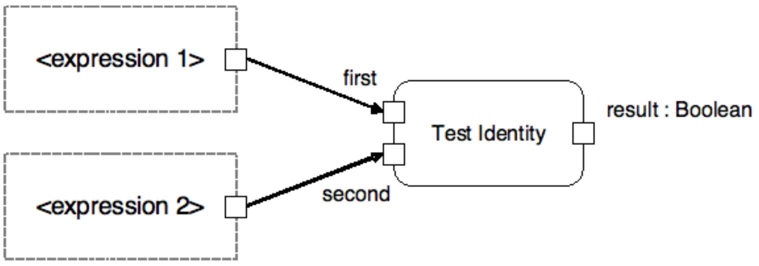
A.5.8 Testing For Equality

Java

```
<expression 1> == <expression 2>
```

- Neither expression may evaluate to null (for testing for null, see A.5.10).
- For UnlimitedNatural values, their integer values must be compared, not the object themselves (see also A.5.2).
- The expressions may not be of type String (for testing string equality, see A.5.9).
- The expressions may not have list types (for more on lists, see A.6).

UML



- An equality test maps to a test identity action. The result output pin of the test identity action becomes the result pin of the mapping.
- The first argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the left sub-expression. The second argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the right sub-expression.
- The expression "<expression 1> != <expression 2>" is mapped as if it was "!(<expression 1> == <expression 2>)."

Notes

- For primitive values, the test identity action tests for equality of value. For object references, it tests the identity of the referent objects.

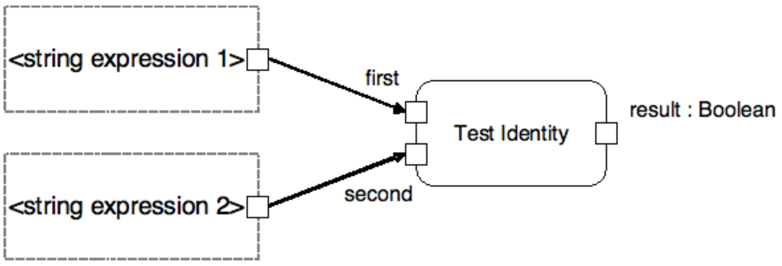
A.5.9 Testing String Equality

Java

```
<string expression 1>.equals(<string expression 2>)
```

- Strings are never tested for equality using "==".

UML



- A string equality test maps to a test identity action. The result output pin of the test identity action becomes the result pin of the mapping.
- The first argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the left sub-expression. The second argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the right sub-expression.

Notes

- In Java String is a class, and testing string values using “==” tests the identity of the string objects being tested, not equality of their values. In UML String is a primitive type, and the test identity action tests for equality of value for strings.

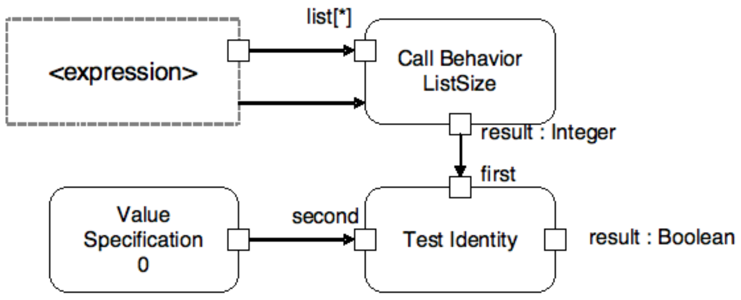
A.5.10 Testing For Null

Java

```
<expression> == null
```

The expression being tested may not have a list type (see A.6).

UML



- A test for null is mapped to a test for whether the result of the mapping of the expression has a list size of zero.
- The result pin of the mapping of the expression is connected by an object flow to the argument pin of a call behavior action for the ListSize behavior (with multiplicity *).
- The call behavior action has a control flow from the action owning the result pin of the mapping of the list expression.
- The result output pin of the call behavior action is connected by an object flow to the first argument pin of a test identity action. The second argument pin of the test identity action is connected by an object flow to the result pin of a value specification action for the integer value "0". The result output pin of the test identity action becomes the result pin for the mapping.
- The expression “<expression> != null” is mapped as if it was “!(<expression> == null)”.

Notes

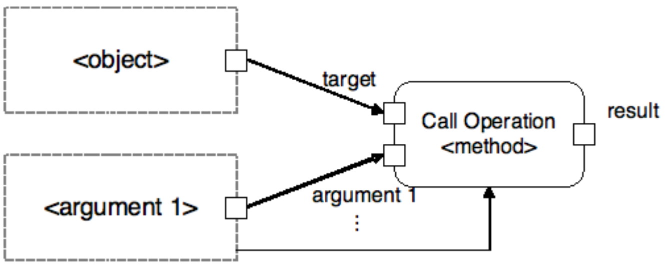
- Java null is used to represent the case of “no value” for a class type with multiplicity [0..1] (see A.5.3).
- The ListSize behavior is provided in the Foundational Model Library (see 9.3.6; see also A.6.6).
- Since the input pin to the call behavior action has a multiplicity lower bound of 0, the control flow is necessary to ensure that the call does not happen before the completion of execution of the mapping of the list expression.

A.5.11 Method Call

Java

```
<object>.<method>(<argument 1>,…)
```

UML



- A method call maps to a call operation action for the operation corresponding to the named method. The result output pin of the call operation action becomes the result pin of mapping. (If the method has a void return type, then there is no result pin.)
- The target input pin of the call operation action is connected by an object flow to the result pin of the mapping of the object expression.
- Each argument input pin (if any) of the call operation action is connected by an object flow to the result pin of the mapping of the corresponding argument expression (in order).
- Unless an argument is of a primitive type, the call operation action has a control flow from the action that owns the result pin of the mapping of the argument expression.

Notes

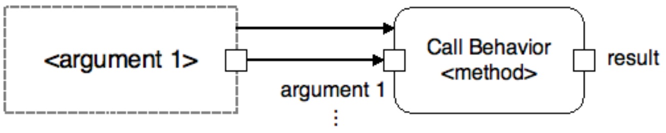
- Since all non-primitive types map to UML types with multiplicity [0..1] or [*] (see A.2), the control flows are necessary to ensure that the call operation action does not start executing before the arguments are computed.

A.5.12 Super Call

Java

```
super.<method>(<argument 1>,…)
```

UML



- A super call maps to a call behavior action for the UML method (the behavior, not the operation) that implements the UML operation corresponding to the Java method in the superclass. The result output pin of the call behavior action becomes the result pin of mapping. (If the method has a void return type, then there is no result pin.)

- Each argument input pin (if any) of the call behavior action is connected by an object flow to the result pin of the mapping of the corresponding argument expression (in order).
- Unless an argument is of a primitive type, the call operation action has a control flow from the action that owns the result pin of the mapping of the argument expression.

Notes

- Since all non-primitive types map to UML types with multiplicity [0..1] or [*] (see A.2), the control flows are necessary to ensure that the call operation action does not start executing before the arguments are computed.
- This is different than the normal mapping of a method call (see A.5.11), but it is not really an exception, since "super" is not actually a proper expression in Java.

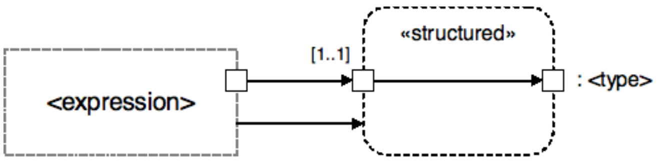
A.5.13 Type Cast (non-primitive)

Java

```
(<type>)<expression>
```

- The expression being cast cannot be of a primitive type.
- The expression being cast cannot be a list.

UML



- A type cast is mapped to a structured activity node that simply copies its input to its output. The input pin of the node is un-typed. The output pin of the node is given the result type of the cast, and it becomes the result pin of the mapping.
- The input pin of the structured activity node is connected by an object flow to the result pin of the mapping of the expression being cast.
- The action that owns the result pin of the mapping of the expression is connected by a control flow to the structured activity node.

Note

- This mapping presumes that the cast is legal. Its behavior is not defined if the result of the expression cannot be cast to the given type.

A.5.14 Type Cast (numeric)

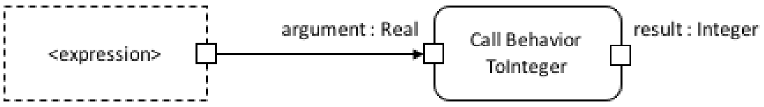
Java

```
(<type>)<expression>
```

- The type must be int or float.

- The expression being cast must be of type int or float.

UML



- A type cast from int to int, float to float, or int to float is mapped as the expression being cast. The cast itself is ignored, other than that the result pin for the expression being mapped is always given the UML type corresponding to the type of the cast.
- A type cast from float to int maps to a call behavior action for the ToInteger behavior. The result output pin becomes the result pin for the mapping. The argument input pin of the call behavior action is connected by an object flow to the result pin of the mapping of the expression being cast.

Note

- In the base semantics, an integer is a kind of real number (see 10.3.1.2), so no actual operation is needed to cast an integer to a real.

A.6 Lists

Classes with names of the form <base type>List are used to represent lists of values of the type <base type>. List classes are mapped to UML multiplicity elements of the form <base type>[*]{ordered, non-unique} (see A.2). Lists of lists are not allowed.

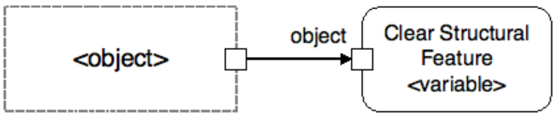
Calls to methods on list classes have special mappings. Calls to the clear, addValue, and removeValue methods map as statements. These methods can only be used on instance variables. A list constructor and calls to the size and get methods map as expressions.

A.6.1 List Clear

Java

```
<object>.<variable>.clear();
```

UML



- A call to the list clear method maps to a clear structural feature action on the attribute corresponding to the list variable.
- The object input pin of the clear structural feature action has an object flow connection to the result pin of the mapping of the object expression.

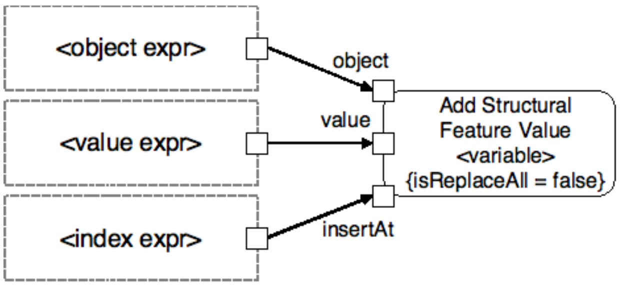
A.6.2 List Add

Java

```
<object expr>.<variable>.addValue(<index expr> - 1, <value expr>)
```

- The value expression must not evaluate to null.

UML


<object expr>
<value expr>
<index expr>
object
value
insertAt
Add Structural Feature Value <variable> {isReplaceAll = false}

- A call to the list add method maps to an add structural feature value action with isReplaceAll = false.
- The object input pin of the add structural feature value action is connected to the result pin of the mapping of the object expression.
- The value input pin of the add structural feature value action is connected to the result pin of the mapping of the value expression.
- The insertAt input pin of the add structural feature value is connected to the result pin of the mapping of the index expression. If the call does not include an index expression, then the insertAt pin is connected to the result output pin of a value specification action for the UnlimitedNatural value "*".

Note

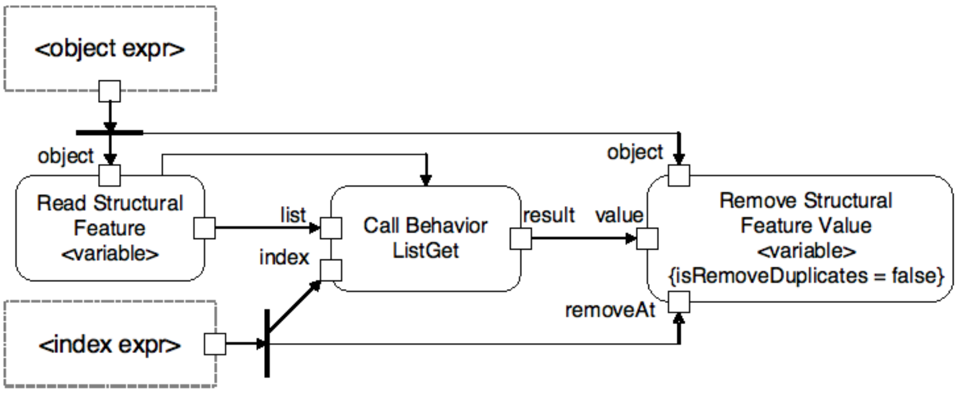
- The Java method indexes from 0, but the add structural feature action indexes from 1.
- Adding a single value to an empty list is an exception to this mapping (see A.6.5).

A.6.3 List Remove

Java

```
<object expr>.<variable>.removeValue(<index expr> - 1)
```

UML



<object expr>
object
Read Structural Feature <variable>
list
index
Call Behavior ListGet
result
value
object
Remove Structural Feature Value <variable> {isRemoveDuplicates = false}
removeAt
<index expr>

- A call to the list add method maps to a remove structural feature value action with isRemoveDuplicates = false.
- The result pin of the mapping of the object expression is connected by an object flow to a fork node, which, in turn, is connected to the object input pin of the remove structural feature value action and the object input pin of a read structural feature action.
- The result output pin of the read structural feature action is connected to the list input pin of a call behavior action calling the ListGet behavior (see A.6.7). There is also a control flow from the read structural feature action to the call behavior action.
- The result pin of the mapping of the index expression is connected by an object flow to a fork node, which, in turn, is connected to the index input pin of the call behavior action and the removeAt pin of the remove structural feature value action.
- The result output pin of the call behavior action is connected by an object flow to the value input pin of the remove structural feature value action.

Note

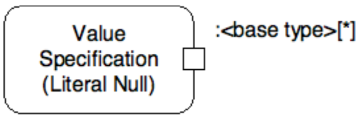
- The Java method indexes from 0, but the remove structural feature action indexes from 1.

A.6.4 Empty List

Java

```
new <base type>List()
```

UML



- A constructor expression for a list type maps to a value specification action for a literal null.

Notes

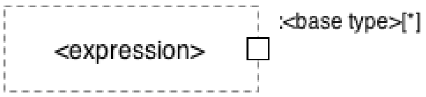
- A value specification for a literal null places no values on its output pin when it executes, corresponding to the 0 cardinality case of the multiplicity [*].
- This is an exception to the normal rule for mapping "addValue" calls (see A.6.2).

A.6.5 List of One Element

Java

```
<base type>List <var> = new <base type>List();
<var>.addValue(<expression>);
```

UML



A list variable initialized by an empty list, immediately followed by adding a single value to that list, maps to the mapping for the expression that is the argument to the "addValue," but with the output pin given multiplicity [*].

Notes

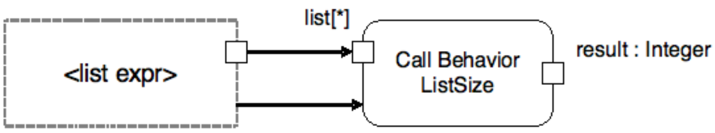
- Since in UML a single element (cardinality 1) conforms to the multiplicity "*", it is not necessary to use an explicit add structural feature value in this case to create the effective mapping of a "list of one element."
- This is an exception to the normal rule for mapping constructor calls (see A.5.5).

A.6.6 List Size

Java

```
<list expr>.size()
```

UML



- A call to the list size method maps to a call behavior action for the ListSize behavior. The result output pin becomes the result pin for the mapping.
- The argument input pin of the call behavior action (with multiplicity *) is connected by an object flow to the result pin of the mapping of the list expression.
- The call behavior action has a control flow from the action owning the result pin of the mapping of the list expression.

Notes

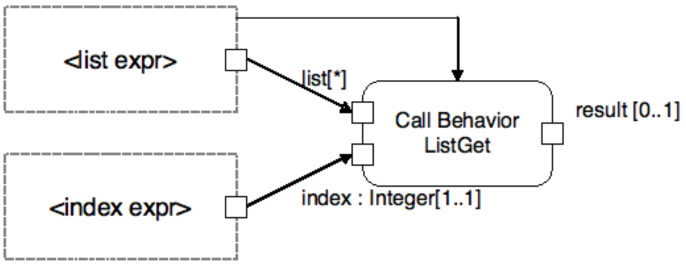
- The ListSize behavior is provided as part of the Foundational Model Library (see 9.3.6). (It can also be defined as an activity, so it does not have to be primitive.)
- Since the input pin to the call behavior action has a multiplicity lower bound of 0, the control flow is necessary to ensure that the call does not happen before the completion of execution of the mapping of the list expression.

A.6.7 List Indexing

Java

```
<list expr>.getValue(<index expr> - 1)
```

UML



- A call to the list get operation maps to a call behavior action for the ListGet behavior. The result output pin of the call behavior action becomes the result pin of the mapping.
- The list argument input pin of the call behavior action is connected to the result pin for the mapping of the list expression.
- The index argument input pin of the call behavior action is connected by an object flow to the result pin for the mapping of the index expression.
- The call behavior action has a control flow from the action that owns the result pin of the mapping of the list expression.

Notes

- The ListGet behavior is provided as part of the Foundational Model Library (see 9.3.6). (It can be defined as an activity and so does not have to be primitive.)
- Since the input pin to the call behavior action has a multiplicity lower bound of 0, the control flow is necessary to ensure that the call does not happen before the completion of execution of the mapping of the list expression.
- The Java method indexes from 0, but the ListGet behavior indexes from 1. If the input index value is less than 1 or greater than the size of the input list, no result is generated.