UNIFIED
MODELING
LANGUAGE

UML Semantics

version 1.0
13 January 1997

ad/97-01-03

RATIONAL
SOFTWARE CORPORATION
2800 San Tomas Expressway
Santa Clara, CA 95051-0951
http://www.rational.com

Contents

1. INTRODUCTION

This document presents the semantics of the Unified Modeling Language (UML). These semantics are specified using a formal textual description together with a metamodel describing the constituents of all well-formed models that may be represented in the UML, using the UML itself. Send any feedback on this document via e-mail to *uml_feedback@rational.com.*

1.1 ORGANIZATION

This formal textual description and metamodel are organized according to the following five packages and associated diagrams:

- Core Concepts
  - Common Elements
  - Common Mechanisms
  - Common Types
- Structural Elements
  - Types, Classes, and Instances
  - Relationships
  - Types
  - Classes
  - Collaborations
- Behavioral Elements
  - State Machines
  - Interactions
- View Elements
  - View Elements
- Standard Elements

The explanation of each diagram is further divided into the following sections:

Description	A statement of the purpose and the scope of the diagram, along with the definition of each relevant part not already defined
Basic Semantics	The meaning of the diagram and its parts
Derived Semantics	The meaning of the diagram and its parts derived from semantics not directly rendered in the given diagram
Standard Elements	A cross reference to the standard stereotypes, tagged values, and constraints that apply to the parts that appear on this diagram, along with any common synonyms for these parts

1.2 Scope

This purpose of this document is to present the complete semantics of the UML in a precise and unambiguous manner. As such, this document is intended for advanced readers, and is not well suited for beginners to learn and understand UML semantics. Indeed, from this document, it may appear that the UML is large and complex. However, much of this seeming complexity stems from the fact that the UML addresses a number of modeling issues that are conceptually simple (for example, the meaning of a type) but formally complex (whole books and theses have been written about the meaning of type). Furthermore, the UML is sufficiently expressive to represent and connect the spectrum of abstraction involved with software development across a wide range of domains, including but not limited to information systems, hard real time systems, web systems, and even certain non-software systems; the UML can handle the modeling of business processes, their corresponding logical and physical software models, and even references to their implementations. Rest assured, the UML is not that complex: the UML is built from a small number of concepts applied consistently across a number of modeling problems.

For most users, the metamodel is invisible, as it should be. However, it is still critical to a select audience, because it facilitates the communication of the precise semantics of the UML. This audience includes members of standards organizations, tool builders, authors, and trainers. It is important to note that the metamodel supports extensibility of the UML so that it may adapt to future advances in object-oriented analysis and design. As such, this metamodel provides UML developers with a stable architecture for trying new modeling problems: if one can model a complex problem easily within the context of the metamodel, then the validity of the metamodel itself is further confirmed.

Thus, this document presents a semantic metamodel, not a tool metamodel. This means that implementations must conform to this semantic metamodel, but may take the liberty of representing the metamodel internally in different ways. By implication, a given implementation may introduce new metamodel classes (for example, to cache values) as well as collapse different metamodel classes into one. These differences notwithstanding, every implementation must conform to the semantics of this metamodel, and must be able to import and export full as well as partial models based upon this semantic metamodel.

1.3 Conventions

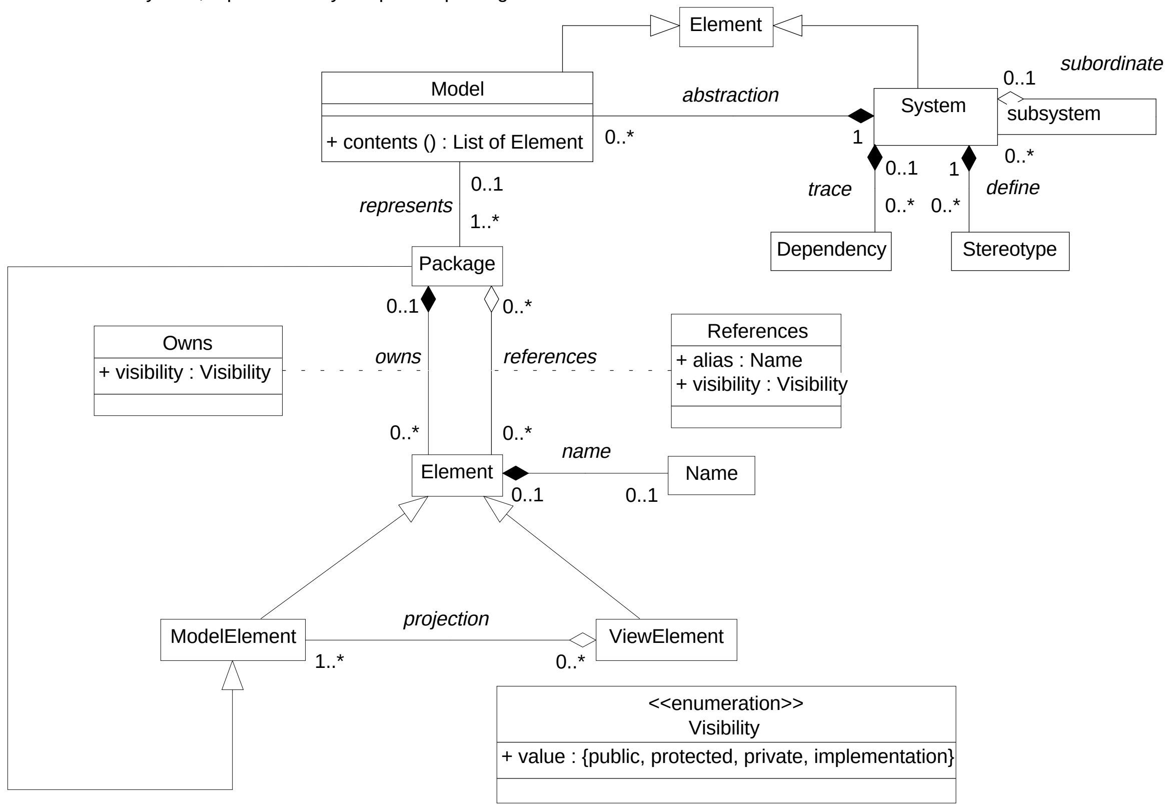
This document uses the UML notation and semantics to describe UML semantics. Although this is a bit of a circular description (to understand the description of the UML semantics, you must understand UML semantics), understanding this document is made simple by the fact that only a subset of the UML is needed to describe UML semantics. Specifically, UML semantics are described in about a dozen class diagrams, mostly consisting of classes (including attributes and operations), associations (including aggregation), and packages. Most classes appear in just one diagram, but some classes (for example Type) appear in several. In general, each appearance of a class in a class diagram presents only those attributes and operations that are relevant to that diagram; the complete interface of a class must be constructed from each appearance of that class across all diagrams.

This document provides a formal description of UML semantics, but is pragmatic about how far it takes this formalism. For example, the description of UML semantics includes phrases such as “The responsibility of X is …” and “X is a Y.” In each of these cases, the usual English meaning is assumed, although a deeply formal description would demand a specification of these semantics of even these simple phrases. As much as possible, the description of UML semantics avoids arcane terms. There are two exceptions to this principle: the use of the terms “manifest” and “face.” In this context, “manifest” means “to make explicit,” and “face” refers to the most significant or prominent surface of an element, especially one used for interaction or communication. The opposite of manifest is implicit.

Thus, the metamodel for the UML is described in a combination of natural language (English) text and class diagrams, written using the UML itself. We recognize that there are theoretical limits to what one can express about a metamodel using the metamodel itself. However, our experience suggests that this combination strikes a reasonable balance between expressiveness and readability for the intended audience.

2. CORE CONCEPTS: COMMON ELEMENTS

Elements are the basic building blocks of the UML. Elements include model elements (which are abstractions drawn from the system being modeled) as well as view elements (which are textual and graphical projections of these model elements). Elements may be organized into packages, which own and reference elements. A model is a semantically closed abstraction of a system, represented by a top-most package.



2.1 DESCRIPTION

This diagram describes the basic building blocks of the UML, and includes the following metamodel classes:

Dependency	Dependency is a unidirectional using relationship from a source (or sources) to a target (or targets).
Element	An element is an atomic constituent of a model.
Model	A model is a semantically closed abstraction of a system.
ModelElement	A model element is an abstraction drawn from the system being modeled.
Name	A name is a string.

Owns	Owns is a composite aggregation of a package to a collection of elements. A package owns elements. Visibility is how the associated element is seen from outside its enclosing package.
Package	A package is a general purpose mechanism for organizing elements into semanically related groups
References	References is a shared aggregation of a package to a collection of elements. A package references elements. Alias is a new name and visibility is the new visibility for the element in the context of its reference.
Stereotype	A stereotype is the classification of an element. A stereotype has semantic impact. Certain stereotypes are predefined in the UML; others may be user defined.
System	A collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibily from different viewpoints.
ViewElement	A view element is a textual and graphical projection of a collection of model elements.
Visibility	Visibility is an enumeration whose value (public, protected, private, or implementation) denotes how the element to which it refers is seen outside its enclosing name space.

This diagram also introduces the following relationships:

abstraction	Abstraction is a composite aggregation of a system to a collection of models. A model is an abstraction of a system.
define	Define is a composite aggregation of a system to a collection of stereotypes A system defines stereotypes.
generalization	Model is a subtype of element.
	Model element is a subtype of element.
	Package is a subtype of model element.
	System is a subtype of element.
	View element is a subtype of element.
name	Name is a composite aggregation of an element to a name. A name is the name of an element.
owns	Owns is a composite aggregation of a package to a collection of elements. A package owns elements.

projection	Projection is a shared aggregation of a view element to a collection of model element. A projection provides both the placement and the style of presentation of each model element within a view element
references	References is a shared aggregation of a package to a collection of elements. A package references elements.
represents	Represents is the association of a model to a top-most package. A top-most package represents a model.
subordinate	Subordinate is a shared aggregation of a system to a collection of systems. The subordinate systems are the subsystems of the system.
trace	Trace is a composite aggregation of a system to a collection of dependencies, each of which represents a trace among elements in different models. A dependency traces from element to element.

2.2 BASIC SEMANTICS

Element is the abstract base class for most constituents of the UML. The responsibility of Element is to provide an anchor upon which a number of common mechanisms may be attached. Each Element instance may have associated with it no more than one Name instance, whose purpose is to uniquely name the Element instance in the context of the name space that encloses the Element instance. The default name of an Element instance that has a name is the null name.

Certain Element instances in this metamodel have attributes; all participate in various relationships. For well-formed models, every attribute value of every Element instance must have a well-defined value. For this reason, all attributes must be given a default value as described in this document; some of these values may represent null values. Implementations are free to extend any enumeration type used in the metamodel by adding new enumerations at the end of the predefined ones. Implementations are also free to mark any attribute value or any relationship as unspecified, although such models are considered incomplete and therefore not well-formed.

Element instances are unique, even if they have the same value for their Name instance. This means that, within a Model instance, all Element instances are related by reference, not by name, unless otherwise specified. Thus, changing the value of a Name instance associated with an Element instance does not change any references to the Element instance (although it does change what that Element instance is called). Across Model instances, however, Element instances are related by name. Furthermore, when transferring partial models, relationships at the edge of these partial models are broken by using name matching, not reference matching. In this manner, it is possible to export Element instances without having to transfer the transitive closure of all the other Element instances with which it has relationships.

The responsibility of ModelElement is to representing an abstraction drawn from the system being modeled. The responsibility of ViewElement is to provide a textual and graphical projection of a collection of ModelElement instances. Both ModelElement and ViewElement are abstract subtypes of Element, and in fact the two are the only immediate subtypes of Element. Projection provides the mapping of ModelElement instances to ViewElement instances. Every ModelElement instance may be projected into zero or more ViewElement instances, and every ViewElement instance may be the projection of one or more ModelElement instances.

Package is a subtype of ModelElement. The responsibility of Package is to provide a general purpose grouping mechanism. Collectively, all of the Element instances that are owned or referenced by a Package instance are called its contents. A Package instance defines a name space, meaning that the immediate contents of a Package instance must have names or aliases that are unique for each kind of content. A Package instance has no semantic meaning, meaning that the Package instance only serves to partition a set of Element instances in the context of a Model instance, and that this partitioning disappears in the executable system.

Owns is a composite aggregation of a Package instance over a collection of Element instances. A given Package instance may own zero or more Element instances, and every Element instance is owned by no more than one Package instance. Since Package is a subtype of ModelElement, a Package instance may own or reference other Package instances. Only one Element instance is not owned by a Package instance, namely, the top-most Package instance represented by a Model instance.

The responsibility of Owns is to establish how a given Element instance may be seen by Element instances that are outside its owning Package instance. The visibility attribute of Owns specifies whether the corresponding Element instance is public, protected, private, or implementation. In the absence of any specification, the owned Element instance is considered to have public visibility.

References is a shared aggregation of a Package instance over a collection of Element instances. A given Package instance may reference zero or more Element instances, and every Element instance may be referenced by zero or more Package instances. In this context, reference means that the referenced Element instance is visible in the Package instance that imports it, and may be used just as if it were owned by the importing Package instance. Whereas destroying a Package instance destroys the Element instances it owns, destroying a Package instance does not destroy the Element instances it references (although it does destroy the references).

The responsibilities of References are to provide a mechanism for renaming a referenced Element instance and to establish how a given Element instance may be seen by Element instances that are outside its referencing Package instance. The alias attribute of References specifies a new name for the referenced Element instance, typically to provide a more meaningful name or to resolve a name clash. Alias is a renaming and as such does not introduce a synonym, meaning that the original simple name may not be used directly in the referencing Package instance. The visibility attribute of References specifies whether the corresponding Element instance is public, protected, private, or implementation. A given referenced Element instance may be given a visibility that is equal to or more restrictive than its visibility in the context of the owning or referenced Package instance from which it is referenced. In the absence of any specification, the referenced Element instance is not renamed and is considered to have the same visibility as from the context of the owning or referenced Package instance from which it is referenced.

Visibility is an enumeration. The responsibility of Visibility is to enumerate the degrees to which an Element instance may be seen outside its enclosing name space. The UML defines four degrees of visibility, named in order of least restrictive to most restrictive; a given implementation of the UML may extend the value of Visibility. The default value of a Visibility instance is public.

In the context of a Package instance, an Element instance that is visible is one to which implicit and explicit Relationship instances may be established. The public contents of a Package instance are visible to any Element instance directly outside the Package instance. Additionally, a Package instance creates a wall around its contents whereby nothing outside the Package instance is visible to the contents of the Package instance unless implicitly or explicitly imported.

Implementation visibility means that the corresponding Element instance is not visible outside the Package instance that owns or references it, nor to the contents of any Package instance that imports it, not even to Package instances that have friend Dependency instances to the given Package instance. . Private visibility means that the corresponding Element instance is not visible outside the Package instance that owns or references it, but is visible to the contents of any Package instance that implicitly or explicitly imports that Package instance and that has a friend Dependency instance to that Package instance, where friend is a stereotyped Dependency. In this manner, the friend Dependency instance extends the visibility of an implicit or explicit import of the contents of one Package instance to another. Protected visibility means that the corresponding Element instance is not visible outside the Package instance that owns or references it, but is visible to the contents of any Package instance that has a Generalization instance to that Package instance or that implicitly or explicitly imports that Package instance and that has a friend Dependency instance to that Package instance. In this manner, the Generalization instance establishes an implicit import of the contents of one Package instance to another. Public visibility means that the corresponding Element instance is visible to any Element instance directly outside the Package instance as well as to the contents of any Package instance that has a friend Dependency instance, a Generalization instance, or an explicit import Dependency instance to that Package instance, where import is a stereotyped Dependency.

The contents of a Package instance may include other Package instances, and this nesting establishes an implicit import of the contents of the nested Package instance to the outer Package instance. Thus, the semantics of visibility are transitive. For example, consider a public Package instance P1 and a private Package instance P2 both owned by Package instance P3, and another Package instance P4 which has an import Dependency instance to P3. The public contents of P1 and P2 are not visible to one another (because there exists no implicit or explicit import relationship between the two). The public contents of P1 and P2 are visible to the other direct contents of P3 (because the nesting establishes an implicit import). Furthermore, the public contents of P3 are visible to the contents of P4 (because P4 explicitly imports P3); this includes P1 and its public contents (because visibility is transitive), but not P2 (because it is private relative to P3).

Model is a subtype of Element, representing a semantically closed abstraction of a system. The responsibility of Model is to name an interesting quanta of Element instances that collectively provide an abstraction of a System instance and that are nearly independent of any other quanta. The meaning of “interesting” is entirely conceptual and is relevant only to the stakeholder viewing the given Model instance. A Model instance is semantically closed and its Element instances are nearly independent in the sense that these instances can be understood in isolation, and that the only Relationship instances that may exist across Element instances that are a part of different Model instances are trace Dependency instances, where trace is a stereotyped Dependency. The default name of a Model instance is the null name.

Represents is a shared aggregation of a Model instance to a top-most Package instance. The responsibility of Represents is to connect a Model instance to a single top-most Package instance and transitively all of its contents.

System is a subtype of Element representing a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints. The responsibilities of System are to name a real world domain that is to be abstracted by zero or more Model instances, to hold all of the trace Dependency instances that cut across these Model instances, and to define all of the Stereotype instances that apply to all of the Element instances representing these Model instances. The conceptual boundaries of a System instance are entirely up to the stakeholder viewing the System instance, and thus a System instance from one perspective might be a subsystem from another perspective, denoted as an Element instance in the abstraction of some larger System instance. The default name of a System instance is the null name.

Abstraction is a composite aggregation of a System instance to a collection of Model instances. Every Model instance is the abstraction of exactly one System instance, and every such System instance may be abstracted by zero or more Model instances.

Subordinate is a shared aggregation of a system to a collection of systems. The responsibility of subordinate is to specify the subsystems of a system.

A System instance defines a name space, meaning that the trace Dependency instances held by the System instance must have unique names and that the Stereotype instances defined by the System instance must have unique names. Trace is a composite aggregation of a System instance to a collection of trace Dependency instances; define is a composite aggregation of a System instance to a collection of Stereotype instances.

2.3 DERIVED SEMANTICS

The semantics of Name are described in section 4.

The semantics of Stereotype and Dependency are described in section 3.

The semantics of all ModelElement subtypes are described in sections 5 through 11. The semantics of all ViewElement subtypes and the projection aggregation are described in section 12.

As described in section 3, Element instances may participate in relationships defined by Dependency. Furthermore, Stereotype, TaggedValue, Note, and Constraint instances may be attached to any Element instance. By implication, any ModelElement or ViewElement instance may participate in these Dependency instances and have these parts since ModelElement and ViewElement are both subtypes of Element. As described in section 6, there are additional kinds of relationships that apply to certain subtypes of ModelElement.

Since Package is a subtype of ModelElement, Package instances may participate in Dependency instances. The only meaningful kind of Dependency instances that may involve Package instances are friend, import, and trace Dependency instances. As described in section 6, Package is specified as a subtype of GeneralizableElement (but not as a participant in an Association relationship),and so Package instances may also participate in Generalization instances. The semantics of Package generalization are described in section 6.

Package instances may own and reference Element instances; however, not all kinds of Element instances may be owned or referenced by a Package instance. Specifically, Model and System are subtypes of Element, but no Package instance may own or reference a Model or System instance. In general, a Package instance may own or reference only Type, Relationship, Behavior, and Collaboration instances (including subtypes of each of these metamodel classes). As described in section 5, for example, this encompasses UseCase and Class, both of which are subtypes of Type; thus, for example, a Package instance may often own or reference UseCase instances. A UseCase instance associated with a Package instance must conform with any lower level UseCase instances that might be attached to the contents of the Package instance. Furthermore, UseCase instances at lower levels may be traceable to UseCase instances at higher levels.

As described below, importing is an explicit stereotyped Dependency among Package instances. However, there is no corresponding explicit export relationship. Rather, a Package instance is said to implicitly export all of the visible Element instances it owns or references.

As subtypes of Element, Model and System instances may have Stereotype, TaggedValue, Dependency, Note, and Constraint instances attached to them.

Not every class in the UML metamodel is a subtype of Element. Any part that does not stand alone (for example, Name) is not an Element. Such parts are always reachable in a System instance, because they are always guaranteed to be a part of some Element instance. Furthermore, any part that is outside of a model is not an Element. Thus, the following metamodel classes are not subtypes of Element and may not participate in Dependency relationships nor have attached Stereotype, TaggedValue, Note, or Constraint instances:

ActionExpression	List	References
ActualArgument	Members	Signals
Boolean	Multiplicity	TimeExpression
BooleanExpression	Name	TypeExpression
Concurrency	Nested	Uninterpreted
Expression	Owns	Visibility
FormalParameter	Point	
GeneralizableElement	Projection	

As a Model instance is evolved, it is common for its contents to at times be incomplete and possibly self-inconsistent. Furthermore, it is common to interchange complete Model instances as well as partial Model instances. In such cases, references to distant Element instances (meaning, Element instances that are not part of the transfer but are in some manner used by those parts in the transfer) must be stubbed. How this is done is outside the scope of the core UML semantics.

2.4 STANDARD ELEMENTS

There are six standard stereotypes that apply to the metamodel classes described in this diagram:

Name	Applies to	Semantics
derived	Dependency	A derived dependency is a stereotyped Dependency whose source and target are both elements, usually but not necessarily of the same type. A derived dependency specifies that the source is derived from the target, meaning that the source is not manifest, but rather is implicitly derived from the target.
facade	Package	A facade is a stereotyped Package that only references (and never owns) elements.
friend	Dependency	A friend dependency is a stereotyped Dependency whose source is a package and whose target is a different package. A friend

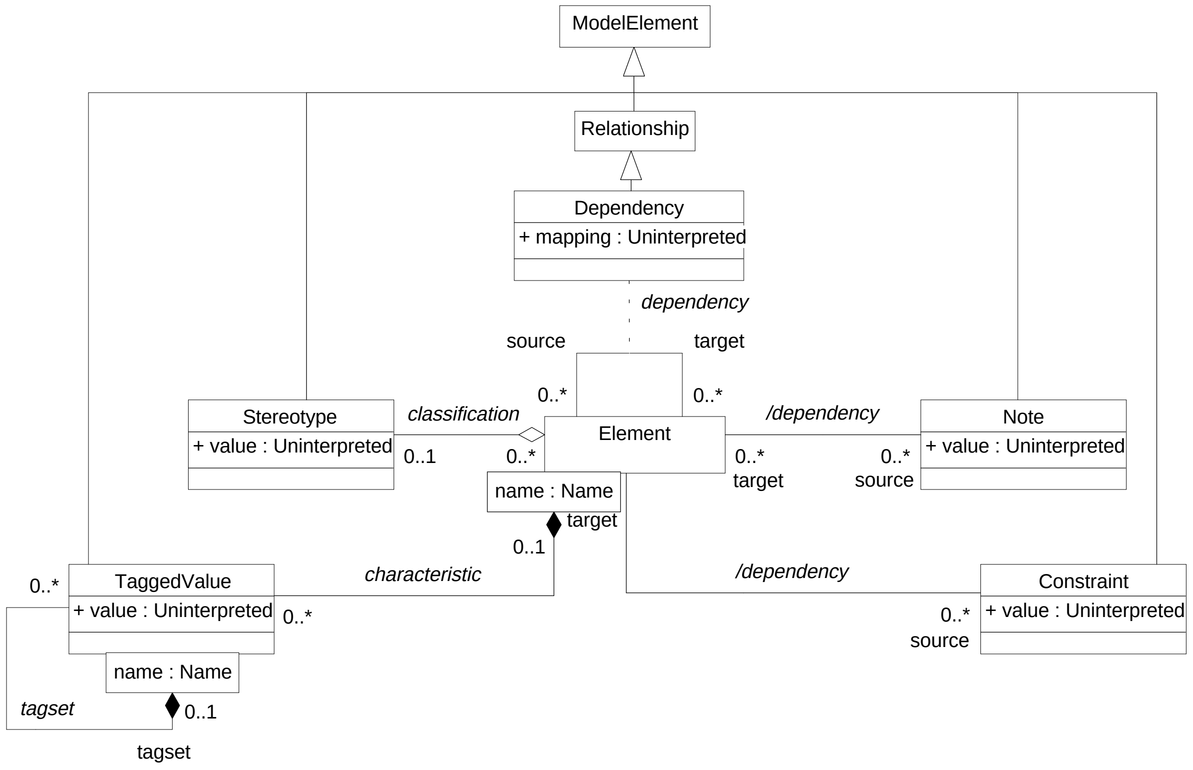
		dependency extends the visibility of an implicit or an explicit import of the contents of one Package instance to another.
import	Dependency	An import dependency is a stereotyped Dependency whose source is a package and whose target is a different package. An import dependency causes the public contents of the target package to be referenceable in the source package.
stub	Package	A stub package is a stereotyped Package representing a package that is incompletely transferred.
trace	Dependency	A trace dependency is a stereotyped Dependency whose source is a model element in one model and whose target is a model element in the same or a different model. A trace dependency is the only kind of relationship that may span model boundaries. A trace dependency indicates that the source conceptually traces back to the target.

There is one standard tagged value that applies to the metamodel classes described in this diagram:

Name	Value	Applies to	Semantics
documentation	String	Element	Documentation is a comment, description, or explanation of the Element instance to which it is attached.

3. CORE CONCEPTS: COMMON MECHANISMS

A small number of common mechanisms exist within the UML, serving to make the UML simple and to give it a sense of conceptual integrity. These common mechanisms include stereotypes, tagged values, notes, constraints, dependency relationships, and the type/instance and type/class dichotomies (the latter of which arespecified in the package Structural Elements).



3.1 DESCRIPTION

This diagram describes the common mechanisms of the UML, and includes the following metamodel classes:

Constraint	A constraint is a condition or restriction attached to an element or a collection of elements. A constraint has semantic impact. Certain constraints are predefined in the UML; others may be user defined.
Dependency	Described in section 2
Element	Described in section 2
ModelElement	Described in section 2
Note	A note is a comment attached to an element or a collection of elements. A note has no semantic impact.

Relationship	A relationship is a semantic connection among elements.
Stereotype	Described in section 2
TaggedValue	A tagged value is a name/value pair denoting a characteristic of an element. A tagged value has semantic impact. Certain tagged values are predefined in the UML; others may be user defined.

This diagram also introduces the following relationships:

dependency	Dependency is a unidirectional using relationship from a source (or sources) to a target (or targets). A source has a dependency on a target.
	A note may be the source of a dependency.
	A constraint may be the source of a dependency
characteristic	Characteristic is a composite aggregation of an element to a collection of tagged values. A tagged value is a characteristic of an element.
classification	Classification is a shared aggregation of an element to zero or one stereotypes. A stereotype is the classification of an element.
generalization	Constraint is a subtype of model element.
	Dependency is a subtype of relationship.
	Note is a subtype of model element.
	Tagged value is a subtype of model element.
	Relationship is a subtype of model element.
	Stereotype is a subtype of model element.
tagset	Tag set is a composite aggregation of a tagged value to a collection of tagged values. A tagged value may be the tag set of a collection of tagged values

3.2 BASIC SEMANTICS

Stereotype is a subtype of ModelElement. The responsibilities of Stereotype are to provide a classification and to optionally establish additional semantics and visual cues for the Element instance to which it is attached. The name of a Stereotype instance is a Name representing the name of the Stereotype instance; its value must not be a null name. The value attribute of Stereotype is Uninterpreted and typically is used to establish new semantics and visual cues for the Element instance to which the Stereotype instance is attached. A Stereotype instance has semantic impact. Certain stereotypes are predefined in the UML; others may be user defined. The semantics of all predefined stereotypes are specified in the UML; the semantics of all user defined stereotypes cannot be enforced by the UML

Stereotype is one of the three extensibility mechanisms of the UML, permitting a modeler to extend the classes of the UML metamodel in controlled ways. Specifically, an Element instance E classified by Stereotype instance S is semantically equivalent to a new metamodel class with the same name as S and whose supertype is the Element instance E. Every predefined stereotype in the UML could have been written explicitly as a new metamodel class whose supertype is the metamodel class to which the stereotype applies. Taken to its natural conclusion, the UML could have been defined by exactly two classes - Thing and Stereotype - with all other metamodel concepts derived as stereotyped Thing instances. This would have been technically correct but practically unapproachable. Therefore, the philosophy taken in the UML is this: all fundamental metamodel concepts that embody sufficiently interesting semantics and that have complex relationships with other concepts are expressed as distinct metamodel classes. Furthermore, any metamodel concept that can be expressed as a simple subtype of these more fundamental metamodel concepts is treated as a stereotype.

Classification is a shared aggregation of an Element instance to no more than one Stereotype instance. The responsibility of Classification is to attach a Stereotype instance to an Element instance. Every Element instance may have at most one Stereotype instance, and every Stereotype instance may be attached to zero or more Element instances.

TaggedValue is a subtype of ModelElement. The responsibility of TaggedValue is to provide a characteristic of the Element instance to which it is attached. The name of a TaggedValue instance is a Name representing the name of the TaggedValue instance; its value must not be a null name. The value attribute of TaggedValue is Uninterpreted. A TaggedValue instance has semantic impact. Certain tagged values are predefined in the UML; others may be user defined. The semantics of all predefined tagged values are specified in the UML; the semantics of all user defined tagged values cannot be enforced by the UML.

TaggedValue is the second of three extensibility mechanisms of the UML, permitting a modeler to extend the attributes of the classes of the UML metamodel in controlled ways. Specifically, an Element instance E with the characteristic TaggedValue instance T is semantically equivalent to the metamodel class E but with a new attribute whose name and type are the name and value of T. Every predefined tagged value in the UML could have been written explicitly as a new attribute in the metamodel class to which the tagged value applies. Taken to its natural conclusion, the UML could have been defined with no attributes but will all characteristics of metamodel classes derived as TaggedValue instances. This too would have been technically correct but practically unapproachable. Therefore, the philosophy taken in the UML is this: all fundamental metamodel class characteristics that embody sufficiently interesting semantics are expressed as distinct attributes.

Characteristic is a composite aggregation of an Element instance to zero or more TaggedValue instances. The responsibility of characteristic is to attach a collection of TaggedValue instances to an Element instance. Every Element instance may have zero or more TaggedValue instances, and every TaggedValue instance may be attached to zero or one Element instances. This aggregation is qualified by the name of the TaggedValue instance, meaning that every TaggedValue instance attached to a given Element instance is uniquely reachable by its name.

Tagset is a composite aggregation of a TaggedValue instance to a collection of TaggedValue instances. The composite TaggedValue instance is known as a tagset, because it represents a set of TaggedValue instances. The responsibility of Tagset is to establish a name for a set of related TaggedValue instances. Every TaggedValue instance is thus either directly the characteristic of an Element instance or a member of a tagset. This is a recursive relationship: TaggedValue instances may define tagsets, and members of these tagsets may themselves define tagsets. This aggregation is qualified by the name of the TaggedValue instance, meaning that every TaggedValue instance in a tagset is uniquely reachable by its name. TaggedValue thus defines a name space, meaning that all TaggedValue instances in a tagset defined by a TaggedValue instance must have unique names.

Relationship is an abstract subtype of ModelElement. The responsibility of Relationship is to establish a semantic connection among Element instances.

Dependency is a subtype of Relationship and is a unidirectional using relationship from a source (or sources) to a target (or targets). The responsibility of Dependency is to name a using relationship wherein the source Element instance (or instances) relies upon the semantics of the target Element instance (or instances). Dependency is thus a many-to-many relationship among types. The name of a Dependency instance is a Name representing the name of the Dependency instance; the name of a Dependency instance is optional, but where it exists it must not be a null name. The mapping attribute of Dependency is Uninterpreted, and is used to record the binding of characteristics of the source from the target. Establishing a Dependency instance between a source and target establishes a unidirectional semantic connection from the source to the target, meaning that if the target is destroyed or its semantics modified, then the source is impacted, the nature of that impact depending upon the specific stereotype of Dependency. For a simple Dependency instance, if all of its sources or all of its targets are destroyed, the Dependency instance is in turn destroyed; there may be no dangling Dependency instances.

Note is a subtype of ModelElement. The responsibility of Note is to provide a comment upon an Element instance or a collection of Element instances. A Note instance has no semantic impact, but may be used by a modeler to attach conceptually interesting information to an Element instance or instances. A Note instance does not have a name. The value attribute of Note is Uninterpreted. Because a Note is a subtype of Element, it may participate in Dependency relationships. A Note instance is attached to other Element instance via a Dependency instance, wherein the Note instance (or instances) is the source (or sources) and the Element instance (or instances) to which it is attached is the target (or targets) of the Dependency instance. Thus, the dependency relationship between Note and Element in the diagram is not manifest, but rather is derived from the dependency relationship that is defined for Element.

Constraint is a subtype of ModelElement. The responsibility of Constraint is to provide a condition or restriction upon an Element instance or a collection of Element instances. A Constraint instance has semantic impact, and may be used by a modeler to attach new semantic constraints to an Element instance or instances. A constraint instance does not have a name. The value attribute of Constraint is Uninterpreted. Because a Constraint is a subtype of Element, it may participate in Dependency relationships. A Constraint instance is attached to other Element instance via a Dependency instance, wherein the Constraint instance (or instances) is the source (or sources) and the Element instance (or instances) to which it is attached is the target (or targets) of the Dependency instance. Thus, the dependency relationship between Constraint and Element in the diagram is not manifest, but rather is derived from the dependency relationship that is defined for Element. A Constraint instance has semantic impact. Certain constraints are predefined in the UML; others may be user defined. The semantics of all predefined constraints are specified in the UML; the semantics of all user defined constraints cannot be enforced by the UML.

Constraint is the third of the three extensibility mechanisms of the UML, permitting a modeler to extend the semantics of the UML in controlled ways. Specifically, an Element instance E with the Constraint instance C is semantically equivalent to the metamodel class E but with new semantics whose value is the value of C. Every semantic concept in the UML could have written explicitly as a new constraint upon the metamodel class to which the constraint value applies. Taken to its natural conclusion, the UML could have been defined with minimal semantics but will all semantics derived as Constraint instances. This would have been technically correct but practically unapproachable. Therefore, the philosophy taken in the UML is this: all fundamental metamodel class semantics that are sufficiently interesting are expressed as distinct semantics.

3.3 DERIVED SEMANTICS

The semantics of Element and ModelElement are described in section 2.

As described in section 2, Element defines a name space. Therefore, all Dependency instances for which a given Element instance is a source or a target must have a unique name. A Dependency instance with no name is always considered to have a unique name, distinct from any other Dependency instance with no name. Similarly, all TaggedValue instances that establish a characteristic of a given Element instance must have a unique name. As described in section 2, Stereotype instances are defined in the context of a System instance, and so all Stereotype instances within a System are already guaranteed to have unique names.

The rule that every Element instance may have at most one Stereotype instance was introduced to simplify the semantics of this extensibility mechanism, because it is based upon single inheritance into the metamodel. As described in section 6, Stereotype instances may participate in Generalization relationships, and thus it is possible for a modeler to establish lattices of Stereotype instances, thereby achieving the effect of multiple inheritance into the metamodel. By restricting Element instances to have at most one Stereotype instance but allowing lattices of Stereotype instances, the semantics of multiple Stereotype instances are made the concern of the metamodel who creates these Stereotype instances instead of the modeler who uses these instances.

TaggedValue instances are commonly used to establish the semantics of mapping a ModelElement instance to a traditional programming language.

There are five concrete subtypes of Relationship. Three of these (Association, Dependency, and Generalization) are structural elements and two of these (Transition and Link) are behavioral elements. The semantics of Association and Generalization are described in section 6. The semantics of Transition are described in section 10. The semantics of Link are described in section 11.

Dependency is a subtype of Element, and so as described in section 2, a Dependency instance may be owned or referenced by a Package instance. A Dependency instance whose sources and targets are all owned by the same Package instance is clearly owned by the same Package instance. A Dependency instance need not be owned by any of the Package instances that own its sources and targets. Thus, a Dependency instance is owned by the smallest name space containing the source and the target. According to the semantics of visibility as described in section 2, a Dependency instance may only be established among sources and targets if and only if those sources and targets are visible. Furthermore, a Dependency instance at least one of whose sources or targets is owned by a different Package instance than the other sources and targets spans the boundaries of its owning Package instance. This introduces further dependencies among the owning Package instances: thus, if all of the Package instances that own the sources or the targets of the Dependency instance are themselves destroyed, then the Dependency instance is in turn destroyed.

The mapping attribute of Dependency plays a role in various stereotypes of Dependency, as described in section 5 and 7.

Visually, a note may be used to project any property of a model. In such cases, a Diagram instance may project a Note instance that is not itself part of a System instance, but rather exists just as a holder for the textual and graphical projection of some other Element instance property. For example, a note might appear in Diagram instance to display a TaggedValue instance or a Constraint instance.

Stereotype, TaggedValue, Dependency, Note, and Constraint are all subtypes of Element, and so themselves may have Stereotype, TaggedValue, Dependency, Note, and Constraint instances attached to them.

The predefined stereotypes, tagged values, and constraints constitute the standard elements of the UML. These standard elements are introduced in subsection 4 of each section and again are summarized in section 13.

3.4 STANDARD ELEMENTS

There are two standard stereotypes that apply to the metamodel classes described in this diagram:

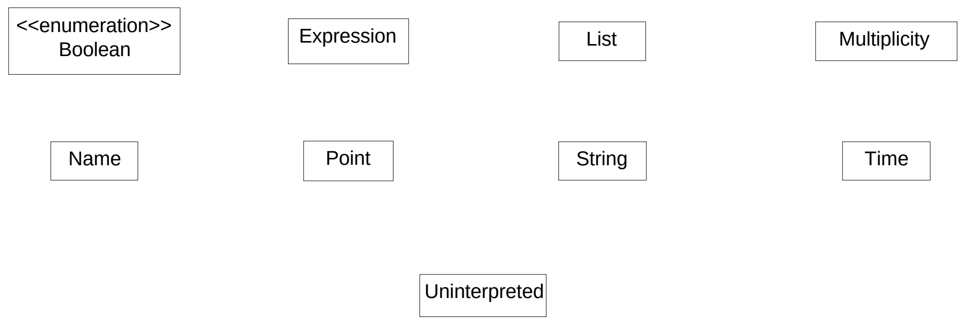
Name	Applies to	Semantics
constraint	Note	A constraint is a stereotyped Note that states a constraint.
requirement	Note	A requirement is a stereotyped Note that states a responsibility or obligation.

There is one synonym that applies to the metamodel classes described in this diagram:

Synonym	Definition
property	A property is any part of an element.

4. CORE CONCEPTS: COMMON TYPES

Common types constitute all of the abstractions that lie at the fringe of the UML and encapsulate it from any implementation-specific usage.



4.1 DESCRIPTION

This diagram describes the common types of the UML, and includes the following metamodel classes:

Boolean	A Boolean is an enumeration whose values are false and true.
Expression	An expression is a string.
List	A list is a container whose parts are ordered and can be indexed.
Multiplicity	A multiplicity is a non-empty set of the non-negative integers extended by a token representing unlimited. Every multiplicity instance has a corresponding string representation.

Name	Described in section 2
Point	A point is an (x, y, z) tuple naming a position in space.
String	A string is a stream of text.
Time	A time is a string representing an absolute or relative moment in time andspace.
Uninterpreted	An uninterpreted is a blob, the meaning of which is domain-specific. Every uninterpreted instance has a corresponding string representation.

4.2 BASIC SEMANTICS

Boolean is an enumeration whose values are false and true. The responsibility of Boolean is to provide a binary value that can be used for attribute type expressions and in guard conditions.

Expression is a string. The responsibility of Expression is to provide a string representing a value. Expression is an abstract class, and every Expression instance evaluates to some value whose type is specified by the Expression subtype. An Expression instance need not be static, but may include Name instances drawn from the scope enclosing the Expression instance. The syntax of a well-formed Expression instance is outside the scope of the UML

List is a container whose parts are ordered and can be indexed. The responsibility of List is to provide a collection of Element instances. The operations applicable to a List instance are outside the scope of the UML.

Multiplicity is a non-empty set of the non-negative integers extended by a token representing unlimited. an open range of non-negative integers. The responsibility of Multiplicity is to specify the range of allowable cardinalities that a set may assume. A Multiplicity instance need not be static, but may include Name instances drawn from the scope enclosing the Multiplicity instance. A Multiplicity that includes a name must have a binding for that name sometime before execution time. The syntax of a Multiplicity instance string representation is specified according to the following production rules:

```
multiplicity ::= [interval | number]{‘,’ multiplicity}
    interval ::= number “..” number
      number ::= non_negative_integer | name | ‘*’
```

Thus, the string representation of a Multiplicity instance is basically a comma separated list of intervals and numbers. An interval is a duple of numbers specifying the closed and inclusive range of non-negative integers from a lower bound to an upper bound, both of which are numbers. A number may be a non-negative integer literal, a Name instance whose value resolves to a non-negative integer, or the star character which denotes an unlimited bound. A multiplicity consisting of a single star denotes the unlimited non-negative integer range. Although not required, the string representation of a Multiplicity instance normally specifies a monotonically increasing and non-overlapping range of non-negative numbers.

Name is a string. The responsibility of Name is to provide an identification by which the Element instance to which it is attached may be called uniquely in the context of its enclosing name space. A Name instance is the simple name of the Element instance to which it is attached. A null Name instance is the null string. All null Name instances are considered unique.

Compound names may be formed through the catenation of simple names, according to the following production rule:

```
     compound_name ::= simple_name {'.' compound_name}
```

A compound name names a path from one Element instance to another. Since the name of an Element instance is guaranteed to be unique only within the context of its enclosing name space, the name of an Element instance may be qualified by the name of the enclosing Package instance that owns or references the Element instance according to the following production rule:

```
    qualified_name ::= qualification "::" simple name
     qualification ::= package_name {"::" qualification}
```

Point is an (x, y, z) tuple naming a position in space. The responsibility of Point is to name a position wherein a ModelElement instance may be projected into a Diagram instance. The origin point of a Diagram instance is the tuple (0, 0, 0), indicating the close in upper left corner. The values of a Point instance tuple are non-negative integers indicating the x, y, and z axis, respectively. The z value of a Point instance tuple may be omitted, in which case a z value of 0 is assumed.

String is a stream of text. The responsibility of string is to provide a name for a stream of text.

Time is a string representing an absolute or relative moment in time/space. The responsibility of Time is to provide a name for a moment in time/space. The syntax of a well-formed Time instance is outside the scope of the UML.

Uninterpreted is a blob, meaning that its value is a binary stream. The responsibility of Uninterpreted is to isolate implementation-dependent properties of the UML. Every Uninterpreted instance has a corresponding string representation, although the syntax of a well-formed Uninterpreted instance is outside the scope of the UML. A null Uninterpreted instance is the null string.

4.3 DERIVED SEMANTICS

None of the classes described in this diagram are subtypes of Element.

There are four concrete subtypes of Expression: ActionExpression, BooleanExpression, TimeExpression, and TypeExpression. The semantics of ActionExpression are described in section 10. The semantics of BooleanExpression and TimeExpression are described in section 10, and the semantics of TypeExpression are described in section 5. An ActionExpression evaluates to an Action instance. A BooleanExpression instance evaluates to a Boolean instance. A TimeExpression instance evaluates to a Time instance. A TypeExpression instance evaluates to a Type instance.

As described in section 7, Type instances may own other Type instances via nesting, and these semantics impact the semantics of qualified names.

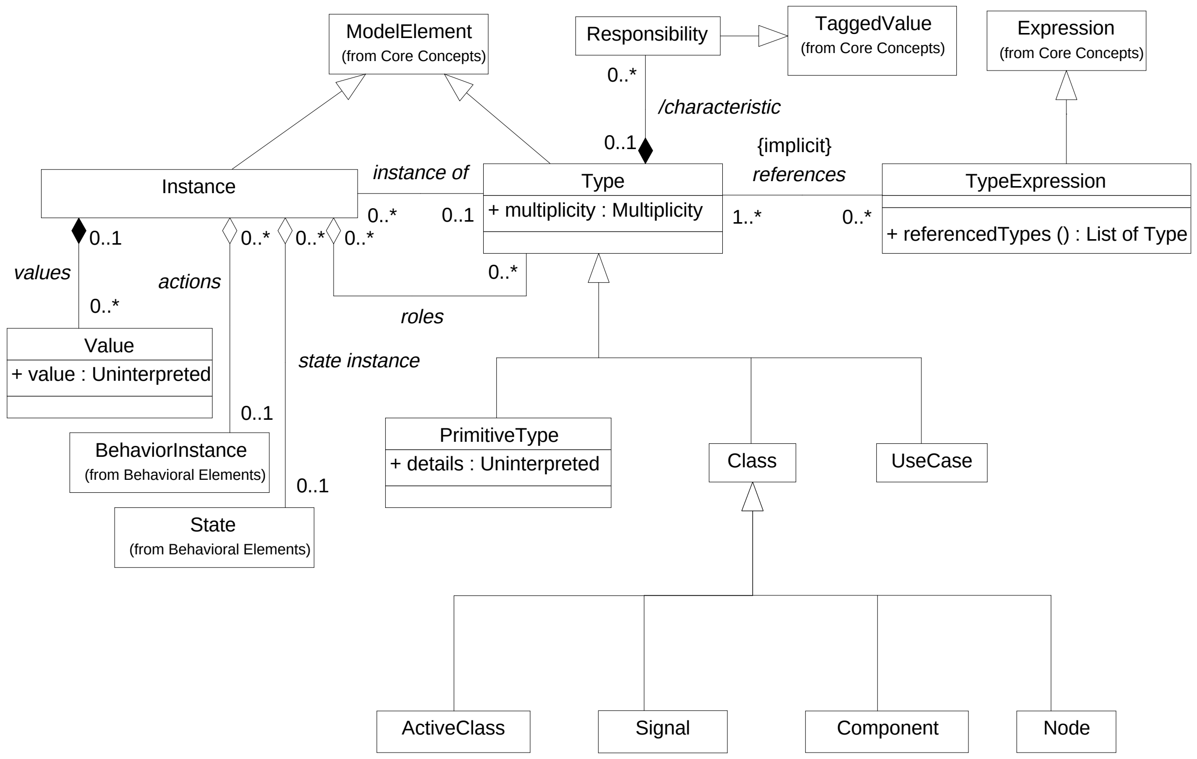
4.4 STANDARD ELEMENTS

There is one synonym that applies to the metamodel classes described in this diagram:

Synonym	Definition
simple name	A simple name is a name.

5. STRUCTURAL ELEMENTS: TYPES, CLASSES, AND INSTANCES

Many model elements exhibit a common essence/manifestation, wherein the type denotes the essence of the abstraction, and the instance denotes a concrete manifestation. There also exists a specification/realization dichotomy, wherein classes and primitive types provide the realization of types.



5.1 DESCRIPTION

This diagram describes the type/instance and the type/class relationships (two of the essence/manifestation and specification/realization dichotomies of the UML, respectively), and includes the following metamodel classes:

ActiveClass	An active class is a class embodying one or more threads.
BehaviorInstance	A behvior instance is a concrete manifestation of a behavior.
Class	A class is the realization of type.
Component	A component is a reusable part that provides the physical packaging of model elements
Expression	Defined in section 4

Instance	An instance is a concrete manifestation of a type.
ModelElement	Defined in section 2
Node	A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well, and upon which components may be deployed.
PrimitiveType	A primitive type is a non-class type, such as an integer or an enumeration.
Responsibility	A responsibility is a contract by or an obligation of the type to which it is attached.
Signal	An signal is a named event.
State	A state is the condition of an instance at a given moment in time/space.
TaggedValue	Described in section 3
Type	A type is the specification of a domain together with behavior applicable to that domain.
TypeExpression	A type expression is an expression that resolves to the reference of one or more types.
UseCase	A use case is a set of sequences of actions a system performs that yields an observable result of value to a particular actor.
Value	A value is the value of an expression.

This diagram also introduces the following relationships:

actions	Actions is a shared association of an instance to its actions. Temporal instances are manifest as a sequence of actions; a behavior instance specifies the actions of an instance.
characteristic	Defined in section 3
generalization	Active class is a subtype of class
	Class is a subtype of type.
	Component is a subtype of class.
	Instance is a subtype of model element.
	Node is a subtype of class.
	Primitive type is a subtype of type.
	Responsibility is a subtype of tagged value.

	Signal is a subtype of class.
	Type is a subtype of model element.
	Type expression is a subtype of expression
	Use case is a subtype of type.
instance of	Instance of is an association between an instance and its type, indicating that the instance is a concrete manifestation of the type. An instance is an instance of a type.
references	References is an association between a type expression and a collection of types, indicating that the type expression references a given type or types. A type expression references one or more types.
roles	Roles is a shared aggregation of an instance to a collection of roles. At a moment in time/space, an instance plays zero or more roles.
state instance	State instance is a shared association of an instance to its state. At a moment in time/space, an instance has a specific state.
values	Values is a composite aggregation of an instance to a collection of values corresponding to the attributes of the associated type. A value is the value of a type's attributes in the context of a specific instance.

5.2 BASIC SEMANTICS

Type is a subtype of ModelElement. The responsibility of Type is to specify a domain together with behavior applicable to that domain. The name of a Type instance is a Name instance representing the name of the Type; its value may not be a null name. The multiplicity attribute of Type is Multiplicity and is used to specify the number of allowable Instance instances of the Type instance within a specific composite. The default value of multiplicity is 0 .. *. A Type instance defines a name space.

Responsibility is a subclass of TaggedValue and thus is a predefined tagged value. The responsibility of Responsibility is to indicate a contract by or an obligation of the Type instance to which it is attached. The value attribute of Responsibility is a string indicating the contract or obligation of the Type instance to which it is attached. A Responsibility instance is attached to a Type instance via a characteristic relationship. This relationship is not manifest, but rather is derived from the characteristic relationship that is defined between TaggedValue and Element.

TypeExpression is a subtype of Expression. The responsibility of TypeExpression is to provide an Expression instance that resolves to the reference of one or more Type instances, where reference denotes a using rather than a defining occurrence of the Type instance. The operation referencedTypes returns a list of Type instance referenced by the TypeExpression instance.

References is an association between a TypeExpression instance and a collection of Type instances, indicating that TypeExpression instance references a collection of Type instances. The responsibility of references is to establish the relationship between a Type instance and the TypeExpression instances in which it is referenced. Every Type instance may be referenced in zero or more TypeExpression instances, and every TypeExpression instance references one or more Type instances. References is an implicit relationship, meaning that it is not manifest but rather is derivable from the value of the TypeExpression instance itself..

PrimitiveType is an abstract subtype of Type. The responsibility of PrimitiveType is to specify a non-class type, such as an integer or an enumeration. The details attribute of PrimitiveType is Uninterpreted and is intended to specify its realization. Implementations of the UML must provide concrete subclasses of PrimitiveType

UseCase is a subtype of Type. The responsibility of UseCase is to specify a set of use case instances, where a use case instance represents a sequence of actions a system performs that yields an observable result of value to a particular actor.

Class is a subtype of Type. The responsibility of Class is to supply the realization of a Type.

The separation of Type and Class constitutes the specification/realization dichotomy of the UML. Whereas a Class instance supplies the realization of a Type instance, a Type instance itself does not provide a realization. In this sense, a Type instance is the interface of a Class instance. This type/class relationship extends to the subtypes of Class. Thus, a given Type instance may be the interface to zero or more Class instances (including its subtypes ActiveClass, Signal, Component, and Node), and a given Class instance (including its subtypes ActiveClass, Signal, Component, and Node) may supply the realization of zero or more Type instances. Since Class is a subtype of Type, a Class instance is itself a Type instance. However, because Type and Class are independent concepts, it is possible to model type and class lattices separately. As described in the following section, refinement is a stereotyped Dependency relationship that specifies the interface/supplier relationship among Type and Class instances.

ActiveClass is a subtype of Class. The responsibility of ActiveClass is to specify a Class instance that embodies one or more threads.

Signal is a stereotyped Class. The responsibility of Signal is to specify a named event.

Component is a subtype of Class. The responsibility of Component is to specify a reusable part that provides the physical packaging of ModelElement instances.

Node is a subtype of Class. The responsibility of Node is to specify a physical part upon which Component instances may be deployed.

Instance is a subtype of ModelElement. The responsibility of Instance is to specify the concrete manifestation of a Type instance. Whereas a Class instance provides the realization of a Type instance, an Instance instance corresponding to a Class instance manifests a Type instance in time/space, meaning that the Instance instance represents an entity that exists in time and space.

The name of an Instance instance is optional, but where it exists it must not be a null name. An Instance instance defines a name space. By implication, the values, actions, state instance, and roles associated with the Instance instance must have unique names according to their kind.

Instance of is an association between an Instance instance and its Type instance. The responsibility of instance of is to specify that the Instance instance is a concrete manifestation of the Type instance. Every Type instance may have zero or more Instance instances, and every Instance instance is the instance of not more than one Type instance. In most cases, every Instance instance is the instance of exactly one Type instance. However, it is possible to specify Instance instances that have no Type, as is often the case in incomplete and/or evolving models.

Roles is a shared aggregation of an Instance instance to a collection of Type instances. The responsibility of roles is to specify the role that the given Instance instance is playing at a moment in time/space, where role in this context means the face or faces that the Instance instance is presenting to its clients. Whereas an Instance instance is always the instance of exactly one Type instance, the roles of an Instance instance may change.

Value is the value of an expression. The responsibility of Value is to reify a value. The value attribute of Value is Uninterpreted, and its semantics are dependent upon the type of the expression for which the Value instance supplies a value.

Values is composite aggregation of an Instance instance to a collection of Value instances. The responsibility of values is to specify the static values of an Instance instance according to values of the Instance instance that correspond to the Attribute instances that are the members of associated Type instance of which the Instance instance is an instance. Every Instance instance has zero or more Value instances and every Value instance is the value of zero or one Instance instances. The number, name, and type of each Value instance that is the value of an Instance instance must match the number, name, and type of the Attribute instances of the associated Type instance of which the Instance instance is an instance.

BehaviorInstance is the concrete manifestation of a behavior. The responsibility of BehaviorInstance is to reify the occurrence of a temporal flow of actions.

Actions is shared association of an instance to its actions. Only instances of UseCase instance have actions. The responsibility of actions is to specify the flow of actions associated with a temporal Instance instance. Every Instance instance is associated with zero or one Behavior instances and every Behavior instance is associated with zero or more Instance instances. The actions of an Instance instance must match with the possible actions of the UseCase instance of which the Instance instance is a match.

State is the condition on an Instance instance at a given moment in time/space. The responsibility of State is to reify a state.

State instance is a shared association of an Instance instance to its state. The responsibility of state instance is to specify the static State instance of an Instance instance. Every Instance instance is associated with zero or one State instances and every State instance is associated with zero or more Instance instances. The state of an Instance instance must match with the possible states of the Type instance of which the Instance instance is an instance.

The separation of Type and Instance constitutes the essence/manifestation dichotomy of the UML, wherein an Instance instance provides the manifestation of a Type instance. This type/instance relationship extends to the subtypes of Type.

5.3 DERIVED SEMANTICS

The semantics of ModelElement are described in section 2.

The semantics of TaggedValue are described in section 3.

The semantics of Expression are described in section 4.

The semantics of Type are described in section 7.

The semantics of BehaviorInstance are described in section 10.

PrimitiveType is a subtype of Type, and so possesses all of the properties of a Type as described in section 7.

A TypeExpression instance introduces a Dependency from the TypeExpression instance to the Type instances it references. Because of the semantics of Name as described in section 2, changing the Name instance associated with a referenced Type instance only impacts the name that the TypeExpression instance sees; it does not replace the Type instance itself. These semantics have important implications for matching the uses of TypeExpression instances with the Type instances they reference. For example, as described in section 7, Attribute instances may be members of Type instances, and Attribute instances have a type attribute whose value is a TypeExpression. If, for example, attribute A has type T, and if the name of T is changed to U, then A is still of the same Type instance, but that Type instance has the new name U.

Class and its subtypes are subtypes of Type, and so possess all of the properties of a Type as described in section 7. The semantics of Class and its subtypes ActiveClass, Component, and Node are described in section 8. The semantics of the Class subtype Signal is described in sections 7, 10, and 11.

ActiveClass, Class, Component, Instance, Node, PrimitiveType, Responsibility, Signal, State, Type, and Value are all subtypes of Element, and so may have Stereotype, TaggedValue, Dependency, Note, and Constraint instances attached to them. Expression and TypeExpression are not subtypes of Element and so may not have these properties.

Since UseCase is a subtype of Type, UseCase instances may participate in Generalization and Association relationships as described in section 6. Among other things, this implies that UseCase instances may be abstract as well as participate in relationships with other Type instances, including actors (which are stereotyped Type instances) and other UseCase instances. Also, as described in section 7, Member instances associated with a Type instance may be specified to have a direction, indicating whether the Member instance is provided or required of the Type instance.

An Instance instance with no name is always considered to have a unique name, distinct from any other Instance instance with no name.

The roles relationship described in this section interacts with the semantics of refinement as well as with the semantics of the role relationship described in section 6. Basically, the Type instances for which another Type (or subtype of Type) instance is the refinement of constitute the static interfaces of the refining Type instance. The role that a given refining Type instance plays in an Association (as described in section 6) must be equal to or a subset of these static interfaces; this is a statement of the static semantics of the refining Type instance and the role that it plays in an Association instance. The roles relationship described in this section must also be equal to or a subset of these static interfaces; this is a statement of the dynamic semantics of the Instance instance. In other words, the interface of a Type instance is static, but may be subsetted in a given context; further, the interface of an Instance instance is dynamic, because at different moments in time, that Instance instance plays a different role in the world.

Between each Type instance and its Instance instances, the values, actions, state instance, and roles of the Instance instance must match the attributes, actions, states, and roles of the associated Type instance. Matching a Value instance to an Attribute instance means that that name and type of both much match. Matching a Behavior instance to a Behavior instance means that the sequence of actions of the actions must be an instance of one of the potential sequence of actions of the UseCase instance (and not more generally a Type instance, since only UseCase instances are temporal). Matching a state instance to a State means the state instance must be an instance of one of the potential states of the Type instance. Matching a role instance to a Type means that the role instances must be one of the potential roles of the Type instance.

Note that that State instance associated with an Instance instance represents an occurrence of the State instance. This manifestation of a State instance is not made explicit, but rather is a consequence of the semantics described in section 10, wherein State is a part of StateMachine which is a kind of Behavior, and Behavior/BehaviorInstance provide an explicit essence/manifestation dichotomy.

A number of different kinds of essence/manifestation pairs appear in the UML. Specifically, Class/Relationship, Association/Link, AssociationRole/LinkRole, Attribute/Instance, Expression/Value, Parameter/Value, Signal/Message, Operation/Message, and Behavior/BehaviorInstance define essence/manifestation pairs that are manifest in the UML As with Type and Instance semantics, these pairs apply to their subtypes as well.

5.4 STANDARD ELEMENTS

There are eight standard stereotypes that apply to the metamodel classes described in this diagram:

Name	Applies to	Semantics
actor	Type	Actor is a stereotyped Type representing an abstraction that lies just outside the system being modeled.
becomes	Dependency	Becomes is a stereotyped Dependency whose source and target are the same instance, but each with potentially different values, state instance, and roles. A becomes dependency from A to A’ means that the instance A becomes A’ (with its possibly new values, state instance, and roles) at a different moment in time/space.
copy	Dependency	Copy is a stereotyped Dependency whose source and target are different instances, but each with the same values, state instance, and roles (but a distinct identity). A copy dependency from A to B means that B is an exact copy of A. Future changes in A are not necessarily reflected in B.
enumeration	PrimitiveType	Enumeration is a stereotyped PrimitiveType. The details of an enumeration specify a domain consisting of a set of identifiers.
instance	Dependency	Instance is a stereotyped Dependency whose source is an instance and whose target is a type. An instance dependency from I to T means that I is an instance of T.
interface	Type	An interface is a stereotyped Type.
refinement	Dependency	Refinement is a stereotyped Dependency whose source is a type, class, collaboration, or method and whose target is a type, collaboration, or operation. The mapping

		attribute is used to match properties of the source to the target. A refinement relationship whose target is a collaboration may only have a collaboration as a source. A refinement whose target is an operation may only have an operation or a method as a source. This relationship specifies that the source is a refinement of the target, meaning that the source maps to the target but with additional information introduced. For example, a class refines a type, meaning that the class conforms to the type but adds additional information (namely, the realization of the type). Similarly, a method refines an operation, meaning that the method conforms to the operation but supplies a realization. As with any dependency, refinement may involve multiple sources and targets. Thus, a single class may be specified as the refinement of multiple types. In the case of a refinement whose source is a type or a class (and thus whose target must be a type), the source is called the supplier and the target is called the interface. Thus, we may say that the refinement of a type by a class means that the class supplies the interface specified by the type. These semantics apply to the subtypes of class as well. For example, we may say that a component supplies the interface specified by a collection of types.
signal	Class	Signal is a stereotyped Class that specifies a named event.

There are two standard tagged values that apply to the metamodel classes described in this diagram:

Name	Value	Applies to	Semantics
persistence	Enumeration	Type Instance Attribute	Persistence is the specification of the permanence of the state of an instance. Persistence is an enumeration specified as {transitory, persistent}. A transitory instance is one whose state is destroyed

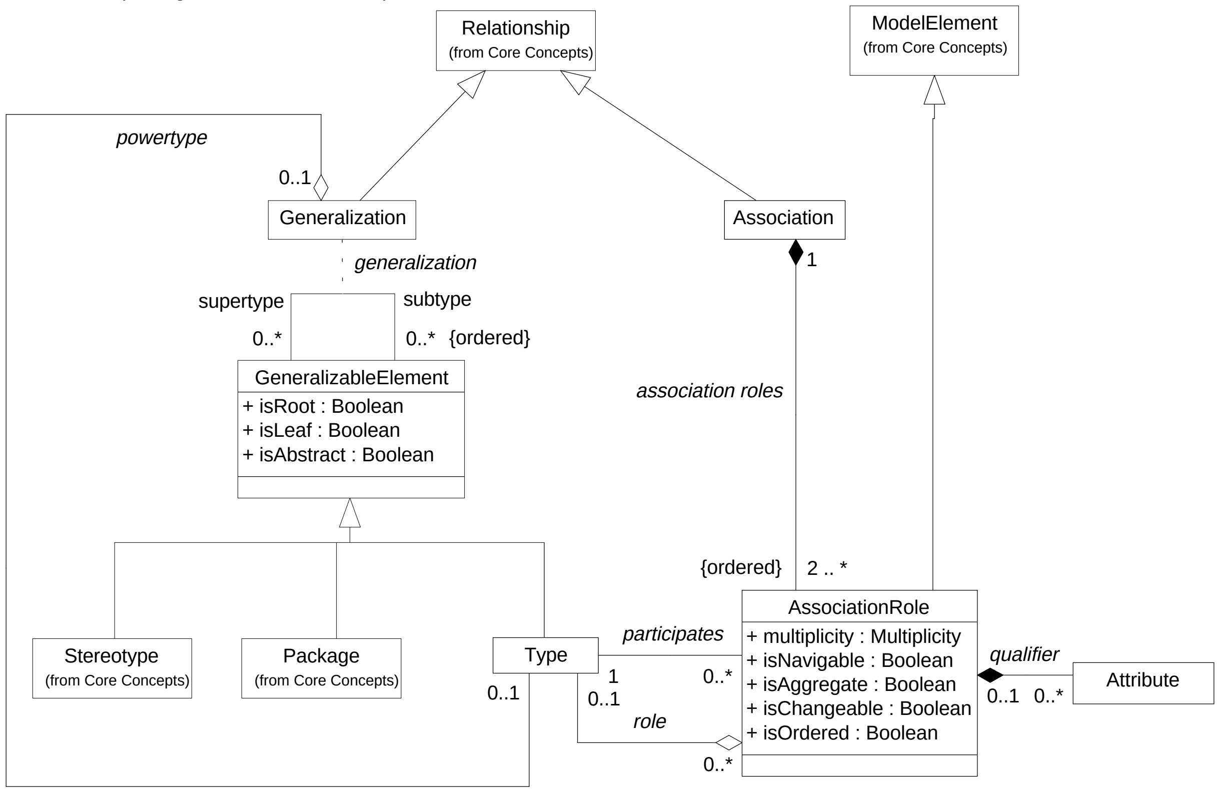
			when the instance is destroyed; a persistent instance is one whose state is not destroyed when the instance is destroyed. The default value of persistence is transitory. Specifying this tagged value on a Type instance constrains the persistence semantics of its instances: all of the instances of a transitory type are transitory, and all of the instances of a persistent type are either transitory or persistent. Specifying this tagged value on an Instance instance states the actual persistence semantics of that instance. Specifying this tagged value on an Attribute instance specializes the persistence property of its owning Type instance.
responsibility	String	Type	A responsibility is a contract by or an obligation of the Type instance to which it is attached.

There are three synonyms that apply to the metamodel classes described in this diagram:

Synonym	Definition
object	An object is an instance of a type (including all subtypes of type).
scenario	A scenario is a defined use case instance.
supplier	A supplier is a type or a class that refines an interface.

6. STRUCTURAL ELEMENTS: RELATIONSHIPS

There are five fundamental kinds of relationships in the UML. One of these - dependency - is a common mechanism that applies to all elements (and is described in the package Core Concepts). Two of these - generalization and association - apply to all types. The remaining two relationship - transition and link - apply to certain behavioral elements (and are described in the package Behavioral Elements).



6.1 DESCRIPTION

This diagram describes the structural relationships of the UML, and includes the following metamodel classes:

Association	An association is a bidirectional semantic connection among instances.
AssociationRole	An association role is the face that a type plays in an association.
Attribute	An attribute is a structural feature of a type. An attribute is semantically equivalent to a composite aggregation with navigation restricted to navigation from the type to the attribute.
GeneralizableElement	A generalizable element is one that may participate in a generalization relationship.

Generalization	Generalization is a unidirectional inheritance relationship, uniting two more more generalizable elements in a supertype/subtype hierarchy, wherein an instance of the subtype is substitutable for an instance of the supertype.
ModelElement	Described in section 2
Package	Described in section 2
Relationship	Described in section 3
Stereotype	Described in section 2
Type	Described in section 5

This diagram also introduces the following relationships:

association roles	Association roles is a composite aggregation of an association to its roles. An association has two or more association roles.
generalization	Generalization is unidirectional inheritance relationship, uniting two more more generalizable elements in a supertype/subtype hierarchy, wherein an instance of the subtype is substitutable for an instance of the supertype. A supertype generalizes a subtype.
generalization	Association is a subtype of relationship.
	Association role is a subtype of model element.
	Generalization is a subtype of relationship.
	Package is a subtype of generalizable element.
	Stereotype is a subtype of generalizable element.
	Type is a subtype of generalizable element.
participates	Participate is an association indicating the role an instance plays in its association with other instances. A type participates in a role.
powertype	Powetype is a composite aggregation of a generalization to one type (the powertype). A type is the powertype of a generalization.
qualifier	Qualifier is a composite aggregation of a role to its attributes (the qualifiers). An attribute qualifies a role.
role	Role is a shared aggregation of a role to zero or one types. A type is the role of a role.

6.2 Basic Semantics

Relationship is an abstract subtype of ModelElement. The responsibility of Relationship is to establish a semantic connection among Element instances. Unless otherwise specified by a subtype, the Name instances associated with all Relationship instances connecting the same Element instance must be unique. Similarly, all well-formed Relationship instances must connect the same or at least two different Element instances: there may be no dangling Relationship instances. Furthermore, destroying the next to the last Element instance connected to a Relationship instance destroys the Relationship instance, since otherwise there would be a dangling Relationship instance.

Generalization is a subtype of Relationship. The responsibility of Generalization is to specify an ordered unidirectional inheritance relationship, wherein an instance of the subtype is substitutable for an instance of the supertype. The Name instance associated with a Generalization instance is called the discriminant of the relationship. For a given supertype, there may be Generalization instances with the same discriminant Name instance, meaning that these identically named relationships partition all of the subtypes of the given supertype into a set named by this discriminant.

GeneralizableElement is an abstract class. The responsibility of GeneralizableElement is to specify an Element instance that may participate in a generalization relationship. A given GeneralizableElement instance may have zero or more supertypes and may be the supertype for zero or more subtypes. A given Element instance may not be a supertype or a subtype of itself. The isRoot attribute of GeneralizableElement specifies if the instance is allowed to have any supertypes; the default value of isRoot is False, meaning that the instance is not the root and hence may have supertypes. The isLeaf attribute of GeneralizableElement specifies if the instance is allowed to have any subtypes; the default value of isLeaf is False, meaning that the instance is not a leaf and hence may have subtypes. The isAbstract attribute of GeneralizableElement specifies if the instance is allowed to have any instances; the default value of isAbstract is False, meaning that the instance is not abstract and hence may have instances.

Stereotype is a subtype of GeneralizableElement. The responsibilities of Stereotype are to provide a classification and to optionally establish additional semantics and visual cues for the Element instance to which it is attached.

Package is a subtype of GeneralizableElement. The responsibility of Package is to provide a general purpose grouping mechanism.

Type is a subtype of GeneralizableElement. The responsibility of Type is to specify a domain together with behavior applicable to that domain.

Powertype is a composite aggregation of a Generalization instance to a Type instance. The responsibility of Powertype is to specify the Type instance that is the powertype of the Generalization instance. Every Generalization instance may have zero or one Type instance as a powertype, and every Type instance may be the powertype of zero or one Generalization instances.

Association is a subtype of Relationship. The responsibility of Association is to specify a bidirectional semantic connection among instances. The Name instance association with the Association instance is the name of the association; for a given Type instance that participates in multiple Association instances, the names of each of these Association instances must be unique.

AssociationRole is a subtype of ModelElement. The responsibility of AssociationRole is to specify the face that a type plays in an association. The Name instance associated with AssociationRole is the name of the association role. Unless otherwise specified, the attributes of an AssociationRole instance are orthogonal. The multiplicity attribute of AssociationRole specifies the number of instances of a Type instance that participate in the Association instance; the default value of multiplicity is 0 .. *. The isNavigable attribute of AssociationRole specifies if the association is navigable to the participating Type instance, where navigable means that given an instance of the Type instance is directly reachable via the Association instance; the default value of isNavigable is True, meaning that the participating Type instance is navigable. Any number of the AssociationRole instances associated with an Association instance may have isNavigable set False. The isAggregate attribute of AssociationRole specifies if the participating Type instance is the whole in a whole/part association. For all the AssociationRole instances associated with an Association instance, at most one AssociationRole instance may have isAggregate set to True, designating the participating Type instance to the be whole of the aggregation and all other participating Type instances to be the parts. When isAggregate is set True for at least one AssociationRole instance that is part of a given Association instance, the value of the multiplicity attribute has semantic implication for the life times of the whole and the part. Specifically, if the multiplicity of the whole is no greater than one, then the whole is said to own the parts, and destroying the whole destroys the parts. If the multiplicity of the whole is greater than one, then the whole is said to share the parts, and destroying the whole does not necessarily destroy the parts. The isChangeable attribute of AssociationRole specifies the mutability of the relationship; the default value of isChangeable is True, meaning that the semantics of the Association instance are preserved even if the instance of the participating Type instance is replaced by a different instance of a Type instance. The isOrdered attribute of AssociationRole applies if the multiplicity of the AssociationRole instance is greater than one, and means that the instances that participate in the Association instance are ordered.

Association roles is an ordered composite aggregation of an Association instance to a collection of AssociationRole instances. The responsibility of association roles is to connect a collection of AssociationRole instances to an Association instance. A given Association instance may have two or more AssociationRole instances, and every AssociationRole instance is a part of exactly one Association instance. The most common Association instance has exactly two AssociationRole instances; Association instances with more than two AssociationRole instances constitute n-ary associations.

An aggregation relationship specifies an Association instance with exactly one AssociationRole instance whose isAggregate attribute is True. Setting this attribute only specifies a whole/part relationship (with the participating Type instance associated with the AssociationRole instance whose isAggregate attribute is True designated as the whole); it says nothing about navigability, ownership, or lifetimes. A composite aggregation is a strong form of aggregation, with a multiplicity of no more than one established for the whole, and isChangeable set to True for the whole. The implication of a composite aggregation is that the whole owns its parts, and that the whole represents a shift in levels of abstraction over the parts. An aggregation relationship that is a multiplicity of greater than one established for the whole is called shared. By implication, a composite aggregation forms a tree of parts, whereas a shared aggregation forms a graph.

Participates is an association indicating the role that an instance of a Type instance plays in an Association instance. The responsibility of participates is to specify the Type instance that participates in a given AssociationRole instance that is part of an Association instance. Every AssociationRole instance has exactly one participating Type instance, and every Type instance may be a participant in zero or more AssociationRole instances.

Role is a shared aggregation of an AssociationRole instance to a collection of Attribute instances. The responsibility of role is to specify how the instances of Type instance are partitioned at one end of an Association instance.

The Attribute instance is said to qualify the Association instance, meaning that across an Association instance with an AssociationRole whose multiplicity attribute specifies greater than one instance of a participating Type instance, the qualifier (or qualifiers) may be used to designate a specific instance of the Type instance.

6.3 DERIVED SEMANTICS

The semantics of ModelElement and Package are described in section 2.

The semantics of Stereotype are described in section 3.

The semantics of Type and Attribute are described in section 7.

If isAbstract is True for a given Stereotype instance, this means that the Stereotype instance many not be the classification of any Element instance.

A Package instance that is the subtype of another Package instance is substitutable for the supertype, meaning that the interface of the subtype Package instance conforms to the interface of the supertype Package instance. In this context, interface means the type of the Package instance, consisting of all of the public Element instances owned by the Package instance and conforms means that the interface of the subtype provides the same structure and behavior of the supertype, although the subtype may provide additional structure and behavior. If isAbstract is True for a given Package instance, this means that the Package instance cannot not stand alone, but must be further refined by a concrete subtype.

GeneralizableElement is not a subtype of Element.

The properties of GeneralizableElement apply to its subtypes as well, and thus the subtypes of Type may also participate in Generalization relationships. As described in section 5, this encompasses Type, Class, ActiveClass, Signal, Component, and Node, all of which may participate in Generalization relationships.

Association is the only subtype of Relationship that specifies a semantic connection among instances; all other relationships (Dependency, Generalization, and Transition) specify connections among types.

N-ary associations are permitted, although there is no manifest ModelElement instance that corresponds to the ViewElement instance that projects the center of the n-ary relationship.

Because Association is a subtype of Element, the stereotyped derived dependency applies, and is typically used to denote association relationships that are not manifest but rather are inherited from an Association instance specified for some supertype of a Type instance that participates in the Association instance.

Association/Link, AssociationRole/LinkRole, and Attribute/Value each form an essence/manifestation pair.

As described in section 5, UseCase is a subtype of Type, and therefore UseCase instances may participate in Generalization and Association relationships. UseCase instances may be the supertype or subtype of other UseCase instances but not of other subtypes of Type. UseCase instances may participate in certain stereotyped Dependency relationships with other UseCase instances as well as other kinds of Type instances, especially actors, which are stereotyped Type instances. UseCase instances may not participate in Association relationships with one another.

6.4 STANDARD ELEMENTS

There are six standard stereotypes that apply to the metamodel classes described in this diagram:

Name	Applies to	Semantics
extends	Generalization	Extends is a stereotyped Generalization whose source and target must both be use cases or types. representing that the behavior of the source use case (or type) extends the behavior of the target use case (or type).
powertype	Dependency	Powertype is a stereotyped Dependency or Type. A powertype Dependency is one whose

	Type	source is a generalization and whose target is a type, specifying that the target is the powertype of the source. A powertype Type is one that represents a type that is only a powertype of a generalization.
role	Dependency	Role is a stereotyped Dependency, whose source is a type and whose target is an association role. A role dependency specifies that the role of the association role is the source target. This role type must be one of the roles of the type that participates in the association role.
subclass	Generalization	Subclass is a stereotyped Generalization, specifying that the subtype is a subclass (but not a subtype) of the supertype. In this context, subclassing means that the subtype inherits the structure and behavior of the supertype, but the subtype is not a type of the supertype.
subtype	Generalization	Subtype is a stereotyped Generalization, specifying that the subtype is a subtype of the supertype. In this context, subtyping means that the subtype inherits the structure and behavior of the supertype, and that the subtype is a type of the supertype.
uses	Dependency	Uses is a stereotyped Dependency whose source and target must both be use cases, representing that the source use cases also includes the behavior of the target use case.

There are seven standard constraints that apply to the metamodel classes described in this diagram:

Name	Applies to	Semantics
complete	Generalization	Complete is a constraint applied to a collection of generalization relationships, specifying that all subtypes have been specified (although so may be elided) and that no additional subtypes are permitted.
disjoint	Generalization	Disjoint is a constraint applied to a collection of generalization relationships, specifying that instances may have no more than one of the

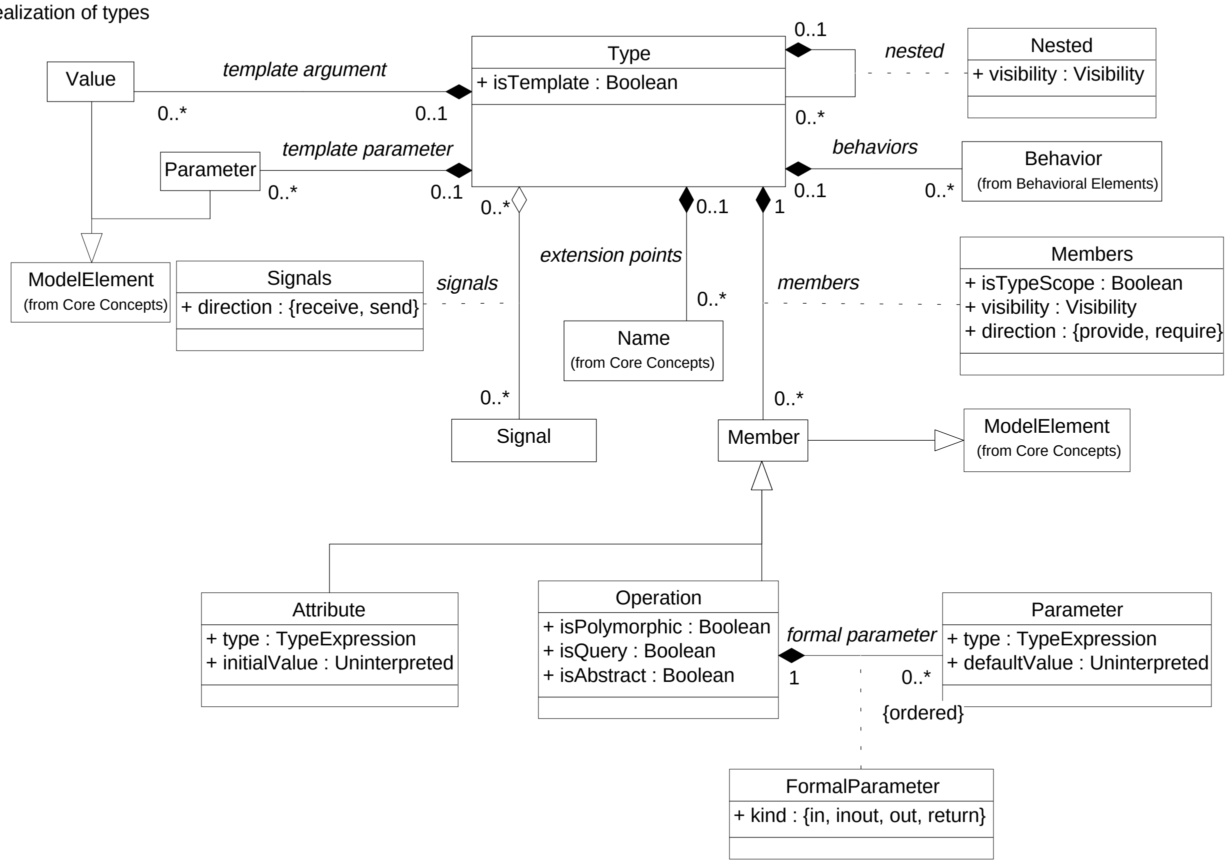
		given subtypes as a type of the instance. This is the default semantics of generalization, meaning that unless otherwise specified as overlapping, all instances may have only one of the subtypes as a type of the instance.
implicit	Association	Implicit is a constraint applied to an association, specifying that the association is not manifest, but rather is only conceptual.
incomplete	Generalization	Incomplete is a constraint applied to a collection of generalization relationships, specifying that not all subtypes have been specified (even if some are elided or are not part of the model) and that additional subtypes are permitted. This is the default semantics of generalization, meaning that unless otherwise specified as complete, all supertypes may have additional subtypes.
or	Association	Or is a constraint applied to a collection of instances, specifying that over that collection, only one is manifest for each associated instance. Or is an exclusive (not an inclusive) or constraint.
ordered	AssociationRole	Ordered is a constraint applied to an association role, specifying that the order of the participants is semantically important.
overlapping	Generalization	Overlapping is a constraint applied to a collection of generalization relationships, specifying that instances may have more than one of the given subtypes as a type of the instance.

There are four synonyms that apply to the metamodel classes described in this diagram:

Synonym	Definition
composite	A composite aggregation is a strong form of aggregation, with a multiplicity of no more than one established for the whole, and isChangeable set to False for the whole.
discriminant	The name of a generalization.
inheritance	Inheritance is a generalization relationship.
shared	A shared aggregation is a weak form of aggregation, with a multiplicity of greater than one established for the whole.

7. STRUCTURAL ELEMENTS: TYPES

A type is the specification of a domain together with behavior applicable to that domain.Classes provide the concrete realization of types



7.1 DESCRIPTION

This diagram describes the semantics of types in the UML, and includes the following metamodel classes:

Attribute	Described in section 6
FormalParameter	Formal parameter is a composite aggregation of a signature to a collection o parameters. A parameter is a formal parameter of a signature/operation. Kind specifies the direction of the parameters.
Member	A member is a part of a type denoting either an attribute or an operation.

Members	Members is a composite aggregation of a type to a collection of members. This collection of members are the members of the type. Type scope is the scope of the member relative to the type or the instance. Visibility is how the associated member is seen from the outside of its enclosing type. Direction is an indication of the member being provided or required by the member to which it is attached.
ModelElement	Described in section 2
Name	Described in section 2.
Nested	Nested is a composite aggregation of a type to a collection of types. A type may be nested in another type. Visibility is how the associated type is seen from outside its enclosing type.
Operation	An operation is the public face that a unit of work presents to the world.
Parameter	A parameter is an unbound expression.
Signal	Described in section 5
Signals	Signals is shared aggregation of a type to a collection of signals. This collection of signals are the signals received or sent by the type. Direction is an indication of the signal received or sent by the type to which it is attached.
Type	Described in section 5
Value	Described in section 5

This diagram also introduces the following relationships:

behaviors	Behaviors is a composite aggregation of a type to a collection of behaviors. The behaviors are the behaviors of the type.
extension points	Extension points is a composite aggregation of a type to a collection of names. The names are the extension points of the type.
formal parameter	Formal parameter is a composite aggregation of a signature to a collection o parameters. A parameter is a formal parameter of a signature/operation.
generalization	Attribute is a subtype of Member.
	Member is a subtype of ModelElement.
	Operation is a subtype of Member.

	Parameter is a subtype of ModelElement.
	Value is a subtype of ModelElement.
members	Members is a composite aggregation of a type to a collection of members. These members are the members of the type.
nested	Nested is a composite aggregation of a type to a collection of types. A type may be nested in another type.
signals	Signals is shared aggregation of a type to a collection of signals. This collection of signals are the signals received or sent by the type.
template argument	Template argument is a composite aggregation of a type to a collection of values. A value is a template argument to a parameter of a template type.
template parameter	Template parameter is a composite aggregation of a type to a collection of parameters. A parameter is a template parameter of a template type.

7.2 BASIC SEMANTICS

The responsibility of Type is to specify a domain together with behavior applicable to that domain. The isTemplate attribute of Type is a Boolean specifying if the Type instance is a template or not. A template Type instance is a kind of type that is not manifest (and therefore may not have instances) but rather must first be bound. A template Type instance may have parameters but may not have any arguments, and a non-template Type instance may not have any parameters and may or may not have any arguments (it may have arguments only if the non-template Type instance represents an instantiation of a template Type instance) The default value of isTemplate is False.

Nested is a composite aggregation of a Type instance to a collection of Type instances. The responsibility of Nested is to specify nested Type instance declarations. The visibility attribute of Nested specifies how the associated nested Type instance may be seen from outside its enclosing Type instance. The default value of visibility is public. Every nested Type instance is owned by exactly one Type instance, and every Type instance may have zero or more nested Type instances. Nesting applies recursively, meaning that nested Type instances may themselves have nested Type instances. Since each Type instance defines a name space, the name of each nested Type instance must be unique within the same level of nesting.

Behaviors is a composite aggregation of a Type instance to a collection of Behavior instances. The responsibility of behaviors is to specify the behavior of the Type instance.

Parameter is a subtype of ModelElement and is an unbound expression whose responsibility is to specify the name, type, and default value of a parameter. The name of a Parameter instance is a Name representing the name of the parameter; its value may not be a null name. The type attribute of Parameter is a TypeExpression instance and is used to specify the type of the parameter. The defaultValue attribute of Parameter is Uninterpreted and applies only when there is no match to the given parameters. The type of the defaultValue must be compatible with the type of the Parameter instance.

Template parameter is a composite association from a Type instance to a collection of Parameter instances. Only Type instances whose isTemplate attribute is True may have template parameters. Every Parameter instance in a template parameter is a part of zero or one Type instances, and every Type instance may have zero or more Parameter instances. The Name instances of the Parameter instances become names that are visible to the template Type instance and can be used in the scope of the template in a manner than conforms to the type of the Parameter instance.

Value is a subtype of ModelElement and is a bound value of an expression. The responsibility of Value is to reify a value.

Template argument is a composite aggregation of a Type instance to a collection of Value instances. Only a non-template Type instance may have template arguments, and then only when the Type instance represents an instantiation of a template Type instance. The responsibility of template argument is to provide a collection of Value instances that bind the template parameters of the unbound template Type instance which the non-template Type instance is instantiating. The template argument Value instances of the non-template Type instance must match the template parameter Parameter instances of the template Type instance in order and must conform in type (but not necessarily in name). Furthermore, the template argument Value instances of the non-template Type instance must involve only those Type instance that are visible to the non-template Type instance.

Signal is named event. The responsibility of Signal is to name a potential event representing a significant occurrence in time/space.

Signals is a shared aggregation of a Type instance to a collection of Signal instances. The responsibility of Signals is to specify the Signal instances to which the given Type instance is obligated to respond to or that it sends. The direction attribute of Signals is an enumeration, specifying the direction of the associated Signal instance. The default direction of a Signal instance is receive. A Signal instance that is received by the Type instance is one to which the Type instance is obligated to handle; a Signal instance that is sent by the Type instance is one which the Type instance may send to its clients. A given Type instance may receive or send to zero or more Signal instances, and a given Signal instance may be received or sent by zero or more Type instances.

The UML does not specify the underlying mechanism whereby a Signal instance is broadcast to a Type instance nor how a Type instance receives or sends a Signal instance.

Member is an abstract subclass of ModelElement. The responsibility of Member is to represent an Attribute instance or an Operation instance.

Members is a composite aggregation of a Type instance of a collection of Member instances. The responsibility of Members is to specify the members of a type. A given Type instance may have zero or more Member instances, and every Member instance belongs to exactly one Type instance. The isTypeScope attribute of Members is a Boolean specifying the scope of the associated Member instance. The default value of isTypeScope is False, meaning that the Member instance is instanced scoped. An instance scoped Member instance is one that is unique to the instance of the Type instance, and a type scoped Member instance is one that is shared by all instance of the Type instance. The visibility attribute of Members is a Boolean specifying the visibility of the associated Member relative to the enclosing Type instance. The default value of visibility is public. The direction attribute of Members is an enumeration, specifying the direction of the associated Member instance. The default direction of Members is provide, meaning that the Member instance is one that is declared in the Type instance. A required Member instance is one that the Type instance requires in order to preserve its semantics. Specifying a required Member instance introduces the Member instance to the Type instance, but does not constitute a declaration of the Member instance. Unlike template parameters, the specification of required Member instances does not introduce a template Member instance, but rather is a statement of the semantics of the Type instance's interface, in which the Member instances that it expects to use are specified.

Attribute is a subtype of Member. The responsibility of Attribute is to specify a structural feature of a Type instance. The Name instance associated with an Attribute instance is the name of the attribute; its value must not be the null name. The type attribute of an Attribute instance specifies the type of the attribute, and the initialValue attribute of the Attribute instance specifies its initial value if not otherwise specified or constructed. The type of the initialValue must be compatible with the type of the Attribute instance.

Operation is a subtype of Member. The responsibility of Operation is to specify a behavioral feature of a Type instance. The Name instance associated with an Operation instance is the name of the operation; its value must not be the null name. The isPolymorphic attribute is a Boolean and specifies whether or not the Operation instance is polymorphic; a polymorphic Operation instance is one that a subtype may reintroduce and provide an alternative Method instance, so than when the Operation instance is called, the overridden behavior is carried out. The default value of isPolymorphic is True. In an inheritance lattice, once isPolymorphic is set False, it cannot be set True for the same Operation lower in the lattice. The isQuery attribute is a Boolean and specifies whether or not the Operation instance is a behavior that preserves the state of the instance. The default value of isQuery is False, which means that the semantics of the Operation instance allow the state of the instance of the Type instance to be modified. A value of True means that the semantics of the Operation must guarantee that the state of the instance of the Type instance not be modified. The attribute isAbstract is a Boolean and specifies if the Operation instance has a corresponding realization. The default value of isAbstract is False, meaning that a corresponding realization can exist; a value of True means that a corresponding realization does not and cannot exist. A Type instance that has one or more provided operations for which isAbstract is True represents one that may not directly have any corresponding instances in the real world, although in a Model instance, there may be Instance instances that correspond to an abstract Type instance, representing a prototypical instance of one of its subtypes.

Extension point is a composite aggregation of a Type instance to a collection of Name instances. Every Type instance may have zero or more extension points, and every Name may be the extension point of no more than one Type instance. The Name instances associated with a Type instance as extension points must have unique names.

Formal parameter is an ordered composite association from an Operation instance to a collection of Parameter instances. Every Parameter instance in a formal parameter is a part of zero or one Operation instances, and every Operation instance may have zero or more Parameter instances. The Name instances of the Parameter instances become names that are visible to the Operation instance as parameters and can be used in the scope of the Operation in a manner than conforms to the type of the Parameter instance. The kind attribute of FormalParameter is an enumeration, the responsibility of which is to specify the kind of the formal parameter. In specifies a parameter whose properties can be observed but not modified; out specifies a parameter whose properties cannot be observed but can be modified; inout specifies a parameter whose properties can both be observed as well as modified; return has the same semantics as out, but designates a parameter that may be used in expressions involving the corresponding Operation instance. The default value of kind is inout. A given Operation instance may have any number of Parameter instances of any of these kinds, including return, although most commonly, an Operation instance will have exactly zero or one return Parameter instances.

7.3 DERIVED SEMANTICS

The semantics of Name and ModelElement are described in section 2.

The semantics of Behavior are described in sections 9 and 10.

FormalParameter, Members, Nested, and Signals are not subtypes of ModelElement.

As described in section 5, UseCase is a subtype of Type. Therefore, all of the properties of Type instances described in this section are applicable to UseCase instances. One exception is that UseCase instance may not be nested.

As described in section 5.4, responsibility is a predefined tagged value that applies to Type and that specifies a contract or obligation of the Type instance to which it is attached. During the lifetime of this Type instance, such responsibilities are typically refined and ultimately realized by the members of the Type instance. It is possible to specify an explicit trace from a responsibility to the Member instances that realize it, by introducing a trace Dependency instance whose target is the Responsibility instance and whose sources are the Member instances of the Type instance to which the responsibility is attached.

As described in section 5, a Type instance specifies an interface, which is realized by a Class instance. The separation of Type and Class constitutes the specification/realization dichotomy of the UML. This dichotomy is described in section 5.

As described in section 5.4, there may be refinement Dependency instances whose source is a Type or Class instance and whose target is a Type instance. The target of a refinement specifies an interface, and this interface specifies a role of the source. Collectively, the target Type instances of all the refinement Dependency instances whose source is a given Type instance are called the roles of the Type instance. This concept of roles interacts with the semantics of association roles as described in section 6. For a given Type instance that participates in AssociationRole instances in multiple Association instances, each such AssociationRole instance specifies a role for that Type instance in the form of another (or the same) Type instance that specifies an interface. The complete roles of a Type instance must be a superset of the association roles in which that Type instance participates.

These semantics yield a separation of the specification/realization hierarchy. Specifically, a given Class instance may the refinement of multiple Type instances, representing the roles of the Class instance, and a given Type instance may be refined by multiple Class instances. These roles are the types of the Class instance. If a given Class instance is not the refinement of any Type instance, the type and the class of the Class instance are the same.

The semantics of refinement also interact with the semantics of instances. As described in section 5, and Instance instance may have an association to a collection Type instances, representing the immediate role of that Instance instance. The roles of an Instance instance must conform to all the potential roles of the corresponding Type instance.

As described in section 6.2, Type is a GeneralizableElement and hence Type instances may be specified as abstract.

As described in section 4.2, qualified names are formed from the catenation of package names and qualifications. Nested Type instances are owned by other Type instances, and so introduce the need for a second kind of qualification, as specified by the following production rule:

```
qualification ::= type_name {“::” qualification}
```

As described in section 6, Type instances may participate in Generalization and Association relationships (and as described in section 3, Dependency relationships). The same is true of template Type instances.

The Parameter/Value binding of an instantiation Type instance to a template Type instance represents a kind of essence/manifestation pair. As for the binding of actual parameters to formal parameters of an operation, the binding of a Value instance to a

Parameter instance introduces an Association relationship between the instantiation Type instance and the type associated with the Value instance, in accordance with the manner in which the corresponding Parameter instance in the template Type instance, once all the bindings have been resolved.

The semantics of Type and Attribute persistence are described in section 5.4.

As described in section 5, Signal is a subtype of Class. Thus, as described in section 6, Signal instances are GeneralizableElement instances and may also participate in Association relationships. This means it is possible to model hierarchies of Signal instances. If a Signal instance S is specified as a signal received by a given Type instance, this means that the Type is obligated to respond to instances of S as well as instances of subtypes of S.

Because Signal is a subtype of Class, a Signal instance may have Member instances including Attribute instances (but rarely will it have Operation instances). These Attribute instances are essentially the formal parameters of the Signal instance, and as described in section 10, these formal parameters may be matched to actual parameters in the context of an Action instance.

The semantics of Visibility are described in section 2.

The direction attribute of Signals and Members permits the specification of roles provided by a Type instance as well as roles wanted by a Type instance. A role provided by a Type instance constitutes an obligation of the Type instance to respond to the Signal instances specified as received, and a declaration of the Member instances specified as provided. Similarly, a Type instance may specify a role wanted by the instance. In this context, a wanted role is subset of the full set of the accumulation of all of the association roles opposite the role the given Type instance participates in. Member instances. This permits an evaluation of closure among the Type instance of a model. Specifically, if a given Type instance sends a given Signal instance, some other Type instance can be checked as receiving that same Signal instance. Similarly, a given Type instance may provide certain Member instances, which in turn are required by some other Type instance. The UML does not specify any strong semantics for the binding of providing and wanting roles, other than to permit their specification and to encourage their matching. The rationale for these semantics are that strong matching semantics are possible only in complete, self-consistent, and unchanging models, where as the vast majority of models are by their very nature incomplete, self-inconsistent, and constantly changing.

Since a Type instance defines a name space, the names of all of the Attribute instances declared by a Type instance must be unique. Similarly, the signatures of all of the Operation instances declared by a Type instance must be unique; since for Operations the name space rules of a Type instance apply to signatures and not names, this means that there may be multiple Operation instances declared by a Type instance each having the same Name instance, but must be distinguished by their parameters names and types.

Type/Instance, Parameter/Value, Attribute/Instance, Signal/Message, Operation/Message each form an essence/manifestation pair.

Member is a subtype of ModelElement, and therefore Member instances may have Stereotype, TaggedValue, Note, and Constraint instances attached (as well as participate in Dependency relationships). Two of the more common uses of these properties with Member instances are to provide a categorization of different kinds of members (via a Stereotype instance attached to each Member instance) and to connect Operation instances to their Method instances (via a refinement Dependency) and in turn Method instance to their code bodies (via a Dependency from the Method instance to a Note containing a view into a file Component instance). The semantics of Method and Component are described in section 8.

An attribute is semantically equivalent to a composite aggregation with navigation restricted to navigation from the type to the attribute.

A derived attribute is one that is not manifest but rather is derived from the semantics of other Member instances. A derived attribute is one that is the source of a derived Dependency as described in section 2.4.

As described in section 8, just as Type/Class form an essence/manifestation dichotomy, so do Operation/Method: an Operation instance represents the interface of a unit of work, and a Method instance represents its realization. As described in section 5.4, the relationship between a Method instance and an Operation instance is that of refinement, namely, the Method instance is a refinement of the Operation instance. These semantics yield a separation of the specification/realization hierarchy. A given Method instance may be the refinement of exactly one Operation instance, and one Operation instance may be refined by many different Method instances. If a given Method instance is not the refinement of any Operation instance, the interface and the realization of the unit of work are collapsed into one construct, namely, the Method instance.

As described in section 6, Type instances may participate in Generalization relationships. Type instances that are subtypes of another Type instance inherit all of the properties of their supertypes, including but not limited to the Stereotype, Property, Note, Constraint, Signal, and Member (including Attribute and Operation) instances, as well as corresponding relationships and representations (via Collaboration instances). Subtypes may as usual add new structural properties as well as override behavioral properties (such as for inherited Operation instances and inherited StateMachine instances, the latter of which arise because a Type instance may be represented by a Collaboration instance, which in turn has Behavior instances, which include StateMachine instances). The semantics of inheritance have particular significance for Operation and StateMachine instances. Specifically, an Operation of a supertype Type instance may be overridden by a corresponding Operation instance in a subtype Type instance, if and only if the supertype Operation instance is marked as polymorphic (its isPolymorphic attribute must be True). The Operation instance of the subtype must conform to the signature of the Operation instance of the supertype, but may override the behavior of the supertype's Operation instance. There is not explicit relationship between the Operation instance of a supertype Type instance and its corresponding Operation instances in the subtypes of the Type instance. Rather, these operations are matched by signature: Operation instances with the same signature are considered to match.

The semantics of invoking an instance of a Signal instance or invoking an instance of an Operation instance are described in sections 10 and 11. Operation instances that are marked as polymorphic (specified by setting their isPolymorphic attribute to True) are dispatched via a dynamic lookup.

As described above, a Type instance may have associated Behavior instances; the purpose of these Behavior instances is to specify the behavior of the Type instance as viewed from the outside. However, as described in section 9, both Type and Operation instances may also be represented by a Collaboration instance. For Type instances, this allows the vocabulary of the abstraction to be expressed, thus specifying the behavior of the Type instance as viewed from the inside. Further, as described in sections 10 and 11 respectively, StateMachine and Interaction are both subtypes of Behavior, and hence either or both can be used to describe these semantics. Specifically, instances of StateMachine may be used to specify the dynamic semantics of a Type instance, and instances of Interaction may be use to reflect the dynamic semantics through a series of scenarios that offer prototypical examples of the Type instance's behavior.

In the case of Operation instances, the semantics of the operation may be expressed in a Collaboration which includes both the participants of behavior as well as the dynamic semantics of their collaboration, either specified by StateMachine instances or reflected by Interaction instances.

These semantics have particular importance for UseCase instances. As described in section 5, a UseCase instance specifies a set of scenarios, where a scenario is a sequence of actions. Thus, a UseCase instance (not to be confused with the Instance instance corresponding to the UseCase instance) is represented by a Collaboration instance that in turn has a collection of Behavior instances that specify all of the potential scenarios of the UseCase instance. As described in section 5, its corresponding Instance instance is a scenario, which is a single flow of actions matching this potential flow of actions.

The rules of matching formal parameters to actual parameters is described in section 10. The association of an Operation instance to a Parameter instance introduces an implicit association between the Operation instance and the Type instances referenced in the type of the Parameter instance. Unless otherwise specified, this association has a multiplicity of 0..1 for both roles.

It is possible to specify an explicit send Dependency from an Operation instance to a Signal instance, representing the signals that may be sent during an invocation of the operation. The source of this send Dependency is an Operation instance, and the target is a Signal instance, both of which must be members and signals, respectively, of the same Type instance.

The semantics of extension points interact with the semantics of the extends stereotype as described in section 6. Specifically, the extension points of a Type instance or a UseCase instance (the only two Element instances for which extends applies) define a point that may be extended by another Type or UseCase instance, where in this context extends means that that behavior of an instance may be interrupted, only to be picked up by the Type or UseCase that extends that behavior. Thus, an extension point is nothing more than a label in a sequence of operations associated with a Type or UseCase instance that names a place that may be extended by another Type or UseCase instance.

7.4 Standard Elements

There are five standard stereotypes that apply to the metamodel classes described in this diagram:

Name	Applies to	Semantics
bind	Dependency	Bind is a stereotyped Dependency whose source is an instantiated type or collaboration and whose target is a template type or collaboration. The mapping attribute is used to match properties of the source to the target. A bind dependency specifies that the source is a binding of the target, wherein the unbound template parameters of the target are bound by name to the template arguments of the source.
call	Dependency	Call is a stereotyped Dependency whose source is an operation and whose target is an operation. A call dependency specifies that the source invokes the target operation. A call dependency may connect an operation to any operation that is within scope, including but not limited to required operations, other operations of the type, and other operations associated with different types but that are still visible.
metaclass	Dependency Type	Metaclass is a stereotyped Dependency and a stereotyped Type. A metaclass Dependency is one whose source is a Type and whose target is a metaclass, specifying that the target is the metaclass of the source. A metaclass Type is one that represents the type of a type; by implication, the instance of a metaclass is a type
send	Dependency	Send is a stereotyped Dependency, whose source is an operation and whose target is a

		signal. A send dependency specifies that the source sends the target signal. A send dependency may connect an operation to any signal that is within scope.
utility	Type	Utility is a stereotyped Type, representing a type that has no instances, but rather is a named collection of non-member attributes and operations, each of which are type scoped.

There are six standard tagged values that apply to the metamodel classes described in this diagram:

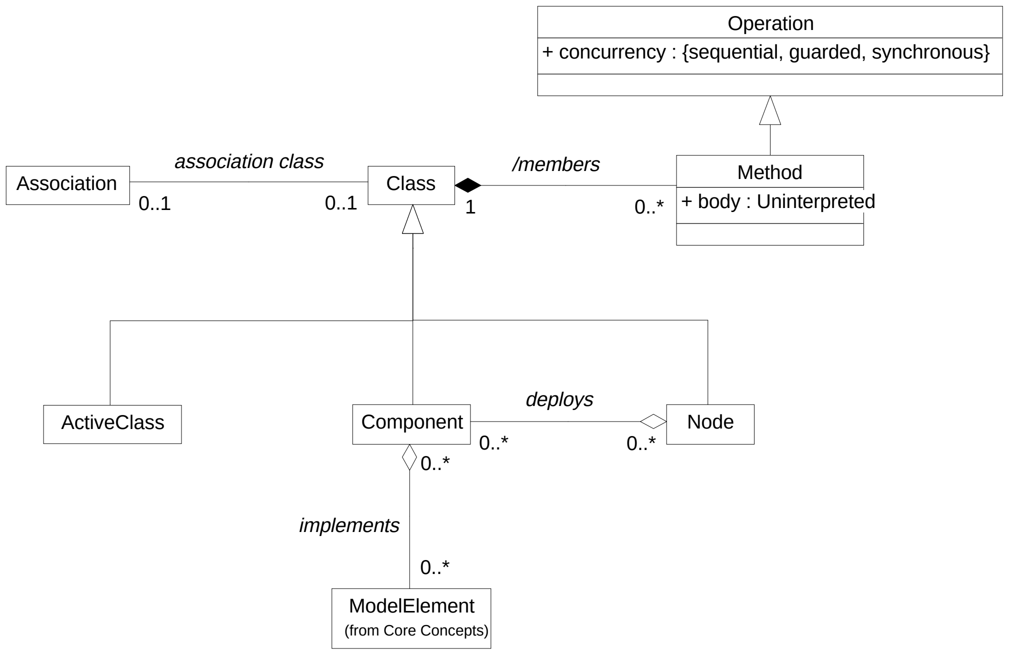
Name	Value	Applies to	Semantics
invariant	Uninterpreted	Type	An invariant is a predicate that specifies properties that must be preserved over the lifetime of any instance of the type.
postcondition	Uninterpreted	Operation	A postcondition is a predicate that specifies that must be held to be true after the completion of the operation.
precondition	Uninterpreted	Operation	A precondition is a predicate that specifies properties that must be held to be true before the operation is invoked.
semantics	Uninterpreted	Type Operation	Semantics is the specification of the meaning of a type or an operation.
space semantics	Uninterpreted	Type Operation	Space semantics is the specification of the meaning of the space complexity of the associated type or operation.
time semantics	Uninterpreted	Type Operation	Time semantics is the specification of the time complexity of the associated type or operation.

There are two synonyms that apply to the metamodel classes described in this diagram:

Synonym	Definition
client	A client is a type that uses an interface.
signature	A signature is the catentation of an operation’s name with the names and types of its parameters. The signature of an operation is essentially its behaviorless interface.

8. STRUCTURAL ELEMENTS: CLASSES

A class is the realization of a type. The UML provides simple classes (Class) together with three specialized classes. One of these specialized classes is behavioral - ActiveClass reifies the concept of thread - and two of these specialized classes are physical - Component is a physical packaging of model elements, and Node is physical part upon which components are deployed.



8.1 DESCRIPTION

This diagram describes the semantics of classes in the UML, and includes the following metamodel classes:

ActiveClass	Described in section 5
Class	Described in section 5
Component	Described in section 5
Method	A method is an elementary quanta of work, providing the realization of an operation.
ModelElement	Described in section 2
Node	Described in section 5
Operation	Described in section 7

Relationship	Described in section 3

This diagram also introduces the following relationships:

association class	Association class is an association between an association and a class. The class reifies the association and thus the class is the association class of the association.
deploys	Deploys is a shared aggregation of a node to a collection of components. A node deploys components.
generalization	ActiveClass is a subtype of Class.
	Component is a subtype of Class.
	Method is a subtype of Operation.
	Node is a subtype of Class.
implements	Implements is a shared aggregation of a component to a collection of model elements. A componet implements model elements.
members	Members is a composite aggregation of a type to a collection of members. These members are the members of the type.

8.2 BASIC SEMANTICS

The responsibility of Class is to provide the realization of a Type. This is the essence of the specification/realization dichotomy in the UML.

Method is a subtype of Operation. The responsibility of Method is to provide the realization of an Operation instance. The body attribute of a Method instance is Uninterpreted, specifying the implementation of the Method instance, typically in a programming language outside the scope of the UML.

The composite aggregation from Class to Method is not manifest, but rather is derived from the members relationship that is defined for Type and Operation. Just as a Type instance may have associated with it a collection of Operation instances, so to may a Class instance have associated with it a collection of Method instances. It is the case that Type instances may only have Operation instances (and not Method instances) as members. Similarly, it is the case that Class instances may only have Method instances (and not Operation instances) as members.

The responsibility of Relationship is to specify a semantic connection among elements.

Association class is an association between an Association instance and a Class instance. The responsibility of association class is to reify an association. The Class instance reifies the association and thus the Class instance is the Association instance class of the relationship. Every Association instance may be associated with zero or one Class instances, and every Class instance may be associated with zero or one Association instances. The Name instance associated with the Relationship instance and the Class instance must match.

Component is a subtype of Class. The responsibility of Component is to specify a reusable part that provides the physical packaging of a collection of ModelElements instances. The semantics of Component represent a shift in levels of abstraction: whereas every ModelElement other than Component (and Node) represents a logical abstraction, Component represents a physical abstraction, meaning an abstraction of a physical implementation of other ModelElement instances.

Implements is a shared aggregation of a Component instance to a collection of ModelElement instances. A Component instance may not implement other Component or Node instances. A Component instance may implement zero or more ModelElement instances, and every ModelElement instance may be implemented by zero or more Component instances.

Node is a subtype of Class. The responsibility of Node is to specify a physical part upon which Component instances may be deployed. In this context, deployed means the allocation of a Component instance to a device upon which the Component instance exists and may act or be acted upon. The semantics of Node represent a shift in levels of abstraction: whereas every ModelElement other than Node (and Component) represents a logical abstraction, Node represents a physical abstraction, meaning an abstraction of the physical distribution of Component instances.

Deploys is a shared aggregation of a Node instance to a collection of Component instances. A Node instance may deploy zero or more Component instances, and every Component instance may be deployed on zero or more Node instances.

The implementation of a ModelElement instance on one or more Component instances and the deployment of a Component instance on one or more Node instances represents the location of that ModelElement or Component. Semantically, the owning Component and Node instance define the location of ModelElement instances and Component instances, respectively.

The responsibility of Operation is to specify a behavioral feature of a Type instance. The concurrency attribute of an Operation instance is an enumeration that specifies the concurrency semantics of the behavior. The default value of concurrency is sequential, meaning that the semantics of the Operation instance are guaranteed only in the presence of a single flow of control. Guarded and Synchronous both guarantee the semantics of the Operation instance in the presence of multiple threads of control, but with different semantics for synchronizing stimuli from these threads. Guarded means that the enclosing Type instance includes a single guard for each instance of the Type instance (and in the case of Operation instances that are type scoped, one other guard for the Type instance itself) such that all guarded stimuli invoked to the instance of the Type instance are sequentialized. Synchronous means that each individual stimuli is sequentialized.

It is possible to have Operation instances with different concurrency semantics as Member instances of the same Type instance. Furthermore, these concurrency semantics apply to all of the subtypes of Type.

ActiveClass is a subtype of Type. The responsibility of ActiveClass is to specify an independent flow of control. Instances of ActiveClass instances are instances just as any other subtype of Type, with the additional semantics that each ActiveClass instance represents the root of an independent and thus concurrent flow of control. The scope of this flow of control is the same as for any Type instance: all instances of ActiveClass instance at the same scope represent peer flows of control, and each such instance may be a whole whose parts are themselves independent flows of control. For composite instances, these flows represent children of the parent ActiveClass instance; for shared instances, these flow represent peers of all flows of control in the same scope as the owner of the shared instance.

8.3 DERIVED SEMANTICS

The semantics of ModelElement are described in section 2.

The semantics of Relationship are described in section 6.

The semantics of Type and Operation are described in section 7.

Class is a subtype of Type, and therefore instances of Class have the same properties as instances of Type, the fundamental difference being that Type instances specify interfaces, whereas Class instances specify the realization of these interfaces. This is the essence of the specification/realization dichotomy in the UML. An implication of this dichotomy is that a Type instance may only have Operation instances as members, whereas Class instances may only have Method instances as members (even though Method is a subtype of Operation and hence Method instances are semantically substitutable). Both Type and Class instances may have Attribute instances as members.

Dynamically, a Class instance may receive Signal instances or Operation instances, but not both.

As described in section 5.4, the relationship between Class instances and the Type instances that it realizes may be explicitly modeled as a refinement Dependency. Furthermore, as described in section 7, if a given Class instance is not the refinement of any Type instance, the type and the class of the Class instance are the same.

The semantics of refinement apply to all of the subtypes of Class. By implication, ActiveClass, Component, and Node instances may be the refinement of Type instances, meaning that these Type instances specify the interface of the abstraction, and the corresponding ActiveClass, Component, and Node instances provide their realization (representing a flow of control, a physical part for packaging, and a physical part for deployment, respectively).

The semantics of refinement apply to Method instances as well: the relationship between a Method instance and the Operation instances it realizes may be explicitly modeled as a refinement Dependency. If a given Method instance is not the refinement of any Operation instance, the operation and the method of the Method instance are the same.

The implementation of a Method instance is typically specified in one of two ways: explicitly, by providing a value for its body attribute, typically in a programming language outside the scope of the UML, or implicitly, derived from the Behavior instance associated with the Method instance through its representation by a corresponding Collaboration instance.

As described in section 9, both Type and Operation instances may be represented by a Collaboration instance. Since Class and Method are both subtypes of Type and Operation, respectively, both Class and Method instances may be represented by a Collaboration instance.

The reification of a Relationship instance as a Class instance impacts the semantics of name spaces. As described in section 6, the Name instance of a Relationship instance must be unique relative to the Element instances that are connected by the Relationship instance. However, the Name instance associated with a Class instance must be unique within its enclosing name space, as described in section 2. Thus, when a Relationship instance is reified as a relationship class, the Name instance associated with the Relationship instance must satisfy stronger semantics for uniqueness.

As described in section 5, Component and Node are both subtypes of Type. Therefore, Component and Node participate in the specification/realization dichotomy of the UML. Specifically, Component and Node instances may both the be the refinement of Type instances, meaning that it is possible to specify the interfaces of a Component or a Node.

As subtypes of Type, Component and Node instances both have the same properties as Type. Most commonly, however, Component and Node instances do not have any signals or members associated with them.

Similarly, as described in section 6, Type instances may participate in Generalization and Association relationships. Since Component and Node are both subtypes of Type, Component and Node instances may also participate in these relationships.

The semantics of threads interact with the semantics of StateMachine instances, as described in section 10. Specifically, each ActiveClass instance has associated with it exactly one event queue, where by events posted to the ActiveClass instance as well as all of the other instances in the scope of that thread are sequentialized. Also as described in section 10, the UML predefines certain Operation instances that are implicit Member instances of every ActiveClass instance; these Operation instances exist to manipulate the ActiveClass instance's event queue.

8.4 STANDARD ELEMENTS

There are eight standard stereotypes that apply to the metamodel classes described in this diagram:

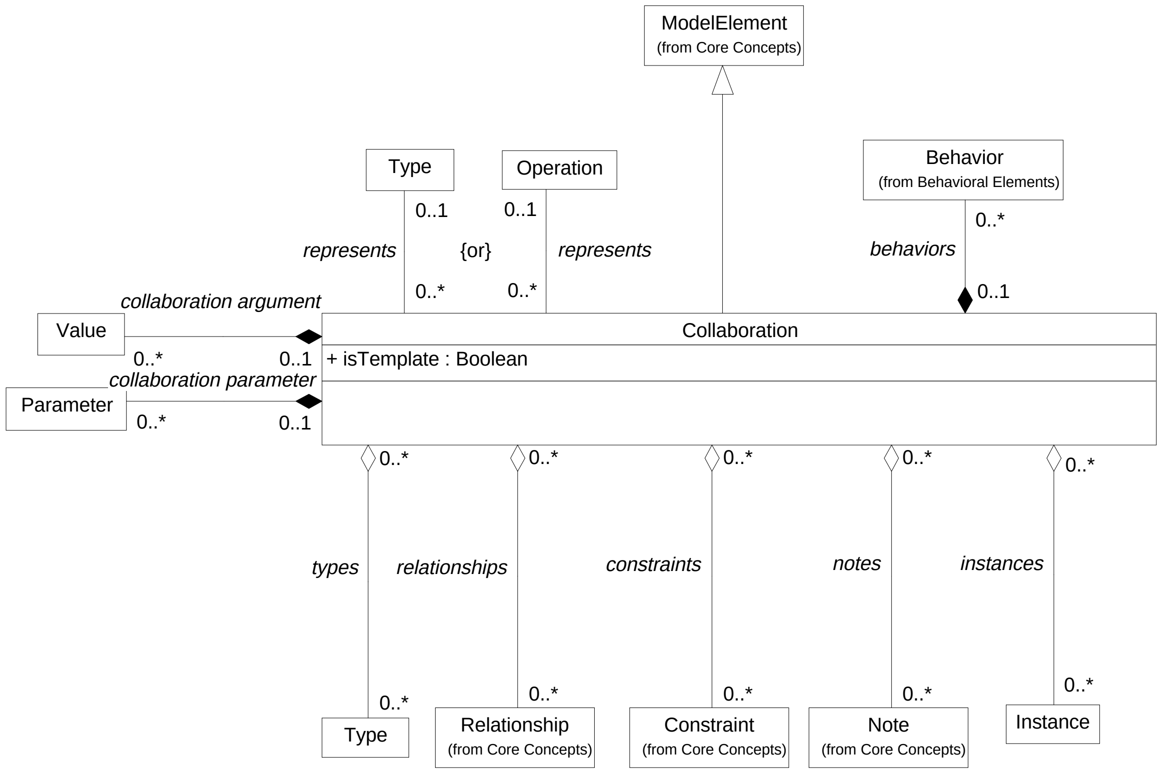
Name	Applies to	Semantics
application	Component	An application is a stereotyped Component representing an executable program.
document	Component	A document is a stereotyped Component representing a document.
file	Component	A file is a stereotyped Component representing a document containing source code.
library	Component	A library is a stereotyped Component representing a static or a dynamic library.
page	Component	A page is a stereotyped Component representing a Web page.
process	ActiveClass	A process is a stereotyped ActiveClass representing a heavy-weight flow of control.
table	Component	A table is a stereotyped Component representing a data base table.
thread	ActiveClass	A thread is a stereotyped ActiveClass representing a light-weight flow of control.

There is one standard tagged value that applies to the metamodel classes described in this diagram:

Name	Value	Applies to	Semantics
location	Component	ModelElement	Location is a derived value of the implementation of a ModelElement instance on collection of Component instances, and of a Component instance on a collection of Node instances.
	Node	Component	

9. STRUCTURAL ELEMENTS: COLLABORATIONS

A collaboration is a mechanism, consisting of structural elements and behavioral elements. Collaborations are an organizational mechanism of the UML, but unlike packages, collaborations have identity and semantic impact. The same element may be a member of more than one collaboration, but the occurrence of an element in each specific collaboration denotes a different society of elements.



9.1 DESCRIPTION

This diagram describes the semantics of collaborations in the UML, and includes the following metamodel classes:

Behavior	A behavior is an observable effect.
Collaboration	A collaboration is a mechanism, consisting of structural elements and behavioral elements.
Constraint	Described in section 3
Instance	Described in section 5
ModelElement	Described in section 2
Note	Described in section 3
Operation	Described in section 7

Parameter	Described in section 7
Relationship	Described in section 3
Type	Described in section 5
Value	Described in section 5

This diagram also introduces the following relationships:

behaviors	Behaviors is a composite aggregation of a collaboration to a collection of behaviors. The behaviors are the behaviors of the collaboration.
collaboration argument	Collaboration argument is a composite aggregation of a collaboration to a collection of values. A value is a collaboration argument to a parameter of a template collaboration.
collaboration parameter	Collaboration parameter is a composite aggregation of a collaboration to a collection of parameters. A parameter is a template parameter of a template collaboration.
constraints	Constraints is a shared aggregation of a collaboration to a collection of constraints. The constraints are the constraints of the collaboration.
generalization	Collaboration is a subtype of ModelElement.
instances	Instances is a shared aggregation of a collaboration to a collection of instances. The instances are the instances in the collaboration.
notes	Notes is a shared aggregation of a collaboration to a collection of notes. The notes are the notes of the collaboration.
relationships	Relationships is a shared aggregation of a collaboration to a collection of relationships. The relationships are the relationships of the collaboration.
represents	Represents is an association between a type or operation and the collaboration it represents. The collaboration represents the type or operation.
types	Types is a shared aggregation of a collaboration to a collection of types. The types are the types of the collaboration.

9.2 BASIC SEMANTICS

Collaboration is a subtype of ModelElement. The responsibility of Collaboration is to specify a mechanism, consisting of structural elements and behavioral elements. A Collaboration instance represents a set of collaborating Type instances, so assembled because that society names a conceptually interesting group.

A Collaboration instance has very different semantics than a Package instance. Both are structuring mechanisms, however, Package instance are only structural whereas Collaboration instances are both structural and behavioral. Furthermore, the contents of a Collaboration instance may transcend Package instance boundaries;

The Name attribute associated with a Collaboration instance represents the name of the Collaboration instance; its name must not be the null name. the isTemplate attribute of Collaboration is a Boolean specifying if the Collaboration instance is a template or not. A template Collaboration is a generative Collaboration that is not manifest but rather must first be instantiated. A template Collaboration instance may have parameters but may not have any arguments, and a non-template Collaboration instance may have parameters and may or may not have any arguments (it may have arguments only if the non-template Collaboration instance represents an instantiation of a template Collaboration instance). The default value of isTemplate is False.

Parameter is an unbound expression. Collaboration parameter is a is a composite aggregation of a collaboration to a collection of parameters. Only Collaboration instances whose isTemplate attribute is True may have template parameters. Every Parameter instance in a collaboration parameter is a part of zero or one Collaboration instances, and every Collaboration instance may have zero or more Parameter instances. The Name instances of the Parameter instances become names that are visible to the template Collaboration instance and can be used in the scope of the template in a manner that conforms to the type of the Parameter instance.

Value is a bound value of an expression. Parameter argument is a composite aggregation of a collaboration to a collection of values. Only a non-template Collaboration instance may have collaboration arguments, and then only when the Collaboration instance represents an instantiation of a template Collaboration instance. The responsibility of collaboration argument is to provide a collection of Value instances that bind the collaboration parameters of the unbound template Collaboration instance which the non-template Collaboration instance is instantiating. The collaboration argument Value instances of the non-template Collaboration instance must match the collaboration parameter Parameter instances of the template Collaboration instance in order and must conform in type (but not necessarily in name). Furthermore, the collaboration argument Value instances of the non-template Collaboration instance must involve only those Type instances that are visible to the non-template Collaboration instance.

A Collaboration instance shares structural elements, including Type, Relationship, Constraint, and Note instances, and owns behavioral elements

The structural dimension of a Collaboration instance arises from the relationships named types, relationships, constraints, notes, and instances, each of which is a shared aggregation from a Collaboration instance to a collection of Type, Relationship, Constraint, Note, and Instance instance, respectively. Each Collaboration instance may share zero or more of these Element instances, and each such Element instance may be shared by zero or more Collaboration instances. By implication, the same Collaboration instance structurally encloses a society of collaborating Type instances, and each such Type instance may be a participant in multiple different Collaboration instances. Because this is a shared aggregation, creating and destroying a Collaboration instance does not impact the lifetime of any of the Element instances it encloses, although destroying one of its shared Element instances removes that Element instance and the transitive closure of its parts from each Collaboration instance of which it is a shared part. A Collaboration instance may encompass any Type, Relationship, Constraint, Note, and Instance instance within the scope that the Collaboration instance is declared.

The dynamic dimension of a Collaboration instance arises from the relationship named behaviors, which is a composite aggregation from a Collaboration instance to a collection of Behavior instances. Every Collaboration instance may have zero or more Behavior instances, and every such Behavior instance is owned by zero or one Collaboration instance.

Represents is an association between a type or operation and the collaboration it represents. Representation is essentially a shift in levels of abstraction. Given a Type or Operation instance and the Collaboration instance that it represents, the Collaboration instance is said to represent the given Type or Operation instance, but viewed from a lower level of abstraction. Every Collaboration instance is the representation of zero or one Type or Operation instance. Every Type and Operation instance may be represented by zero or more Collaboration instances. Collaboration instances may stand alone, meaning that they are not the representation of any Type or Operation.

Collaboration instances have a essence/manifestation dichotomy, but this dichotomy is not made manifest; herein, Collaboration is in effect a type, and its instances are implied. This is not to be confused with templates: a Collaboration instance may be parameterized. Collaboration parameter is a composite aggregation of a collaboration to a collection of parameters. Furthermore, Collaboration argument is a composite aggregation of a collaboration to a collection of values.

9.3 DERIVED SEMANTICS

The semantics of ModelElement are described in section 2.

The semantics of Note and Constraint are described in section 3.

The semantics of Instance are described in section 5.

The semantics of Type, Operation, Value, and Parameter are described in section 7.

The semantics of Behavior are described in sections 9 and 10.

The Behavior instances associated with a Collaboration instance may include both StateMachine instances as well as Interaction instances. As described in section 10, StateMachine instances specify the behavior of a Collaboration instance, meaning that they specify all potential behavior. As described in section 11, Interaction instances reflect the behavior of Collaboration instance, meaning that they record prototypical behaviors. Collaboration instances may include both kinds of Behavior instances to capture their dynamic dimension. The former kind (StateMachine instances) are essentially constructive: they specify all possible paths of behavior, whereas the latter kind (Collaboration instance) are essentially prototypical: each specifies on path of behavior. In this manner, the two views must complement one another: for a given Type instance, the StateMachine must specify behavior that is a superset of that found in a Collaboration instance, and the collection of all such Collaboration instances must conform to that potential behavior.

A Collaboration instance may have zero or more Behavior instances associated with it. Since Behavior is a subtype of Element, Behavior instances may have Stereotype instances associated with them (furthermore, all of the common mechanisms described in section 3 apply to Behavior instances as well). It is common to use Stereotype instances to distinguish different kinds of Behavior instances, such as primary and secondary behaviors.

A Collaboration instance represents a Type as well as any subtype of Type. This means that a Collaboration instance may be the representation of a Type, Class, ActiveClass, Component, Node, and UseCase instance.

Applying a Collaboration instance to a Type, Class, or a subtype of Class instance permits a statement of the semantics of that Element instance. This may be an outside view - specifying the meaning of the Element instance without proscribing its realization - as well as an inside view - specifying the behavior of its realization. In the former case, the structural dimension of the Collaboration instance would contain Element instances that describe the vocabulary of that Type. In the latter case, the structural dimension of the Collaboration instance would contain Element instances that draw from the Member instances and neighbors of that Type. Furthermore, the behavioral dimension of the Collaboration instance permits UseCase instances to be associated with a Type instance (and its subtypes, although in most cases this only applies to Class): as described above, a Collaboration instance may have Type instances and as described in section 5, UseCase is a subtype of Type, hence, a Collaboration instance may have UseCase instances. A UseCase instance associated with a Type instance must conform with any lower level UseCase instances that might be attached to parts of the Type instance.

Applying a Collaboration instance to a UseCase instance permits a statement of the semantics of that UseCase instance. As described in section 5, UseCase is a subtype of Type, and therefore UseCase inherits all of the properties of Type, including the ability to specify a Collaboration instance that represents the UseCase instance. This separation of

UseCase instance and the Collaboration instance that realizes it permits a clear separation of specification and representation of the UseCase semantics. These semantics interact with the semantics of instances as described in section 5. Whereas a UseCase instance may have associated with it Behavior instances that specify the all potential behavior associated with a UseCase instance, an Instance instance representing an instance of a UseCase instance is a scenario, representing a single manifest flow of actions. This, this Instance instance may have an associated Behavior instance representing that flow, drawn from all the potential behavior specified in the UseCase instance.

Applying a Collaboration instance to an Operation instance permits a statement of the semantics of that Operation. As described in section 8, Method is a subtype of Operation, and so these semantics apply to Method instances as well.

As described in section 5.4, refinement is a stereotyped Dependency relationship whose source is a Type, Class, Collaboration, Or Method instance, and whose target is a Type, Collaboration, or Operation instance. The representation relationship from a Type instance to a Collaboration instance and an Operation to a Collaboration instance is a refinement relationship.

Collaboration is a subtype of Element, and so as described in section 3, Collaboration instances may have associated Stereotype, TaggedValue, Dependency, Note, and Constraint instances. Collaboration instances may only participate in Dependency relationships.

9.4 STANDARD ELEMENTS

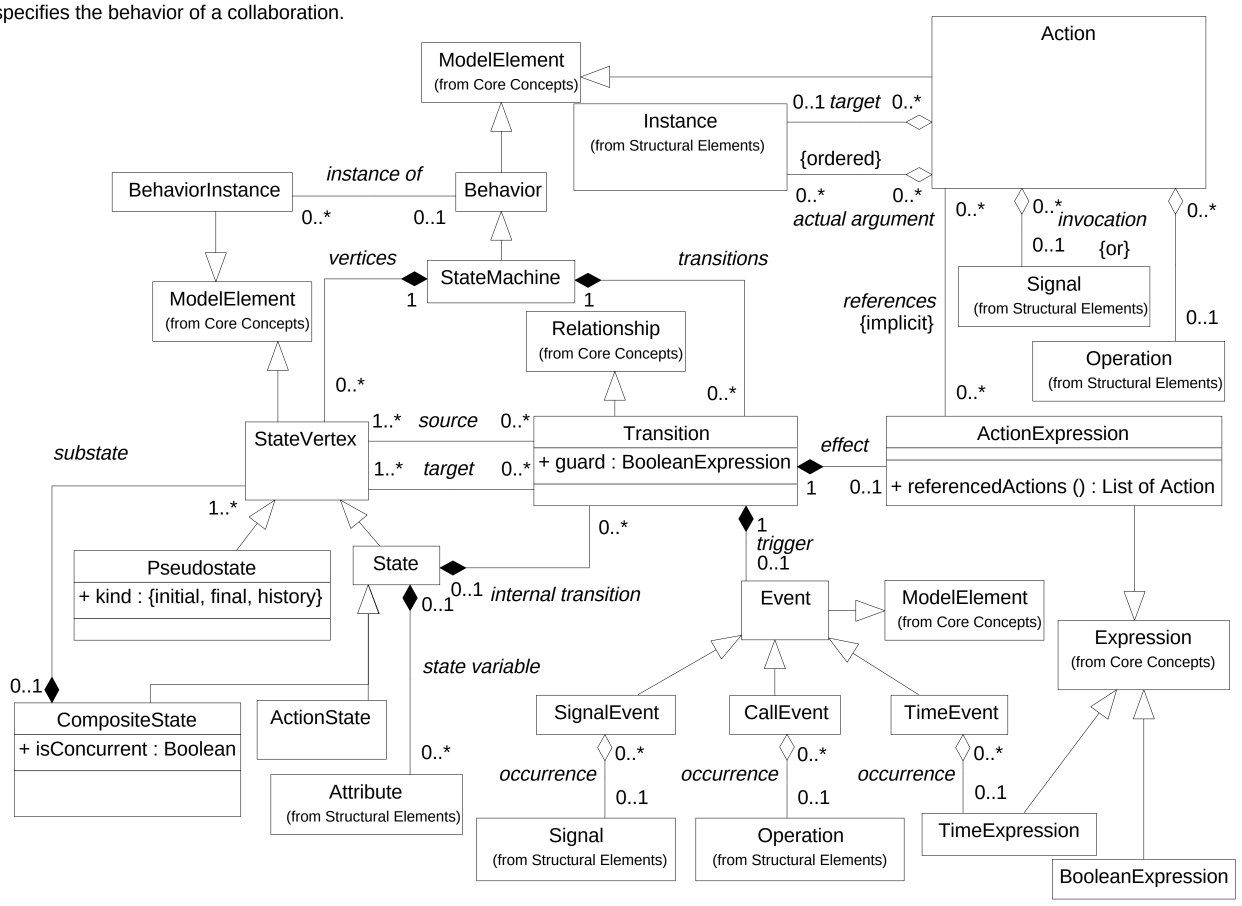
There are two standard stereotypes that apply to the metamodel classes described in this diagram, namely, refinement (described in section 5.4) and bind (described in section 7.4).

There are two synonyms that applies to the metamodel classes described in this diagram:

Synonym	Definition
framework	A framework is a package consisting mainly of patterns.
pattern	A pattern is a template collaboration.

10. BEHAVIORAL ELEMENTS: STATE MACHINES

A state machine is a behavior resulting from operations carried out over a sequence of state changes. A state machine may be viewed from the perspective of states (via state diagrams) or of actions (via activity diagrams). A state machine specifies the behavior of a collaboration.



10.1 DESCRIPTION

This diagram describes the semantics of state machines in the UML, and includes the following metamodel classes:

Action	An action is the invocation of a signal or an operation, representing a computational or algorithmic procedure.
ActionExpression	An action expression is an expression that resolves to a collection of actions.
ActionState	An action state is a state with no substates and exactly one internal transition (on do) with an action expression that resolves to a single operation.
Attribute	Described in section 6

Behavior	A behavior is an observable effect including its results.
BehaviorInstance	Described in section 5.
BooleanExpression	A Boolean expression is an expression that resolves to a Boolean value.
CallEvent	A call event is an event triggered by an operation.
CompositeState	A composite state is a state with substates.
Event	An event is a significant occurrence in time/space.
Expression	Described in section 4
Instance	Described in section 5
ModelElement	Described in section 2
Operation	Described in section 7
Pseudostate	A pseudostate is a non-state vertex. Pseudostates include initial, final, and history connections.
Relationship	Described in section 3
Signal	Described in section 5
SignalEvent	A signal event is an event triggered by a signal.
State	Described in section 5
StateMachine	A state machine is behavior specified as a collection of actions carried out over a sequence of state changes. A state machine specifies the behavior of a collaboration of types.
StateVertex	A state vertex is a source or a target of a transition.
TimeEvent	A time event is an event triggered by the passing of time.
TimeExpression	A time expression is an expression that resolves to a relative or absolute value of time.
Transition	A transition is the passage from one state vertex to another.

This diagram also introduces the following relationships:

actual argument	Actual argument is a shared association of an action to its actual arguments. The instances are the actual arguments of the action.
effect	Effect is a composite aggregation of a transition to at most one action exression.
generalization	Action is a subtype of model element.
	Action expression is a subtype of expression

	Action state is a subtype of state.
	Behavior is a subtype of model element.
	Behavior instance is a subtype of model element.
	Boolean expression is a subtype of expression.
	Call event is a subtype of event.
	Composite state is a subtype of state.
	Event is a subtype of model element.
	Pseudostate is a subtype of state vertex.
	Signal event is a subtype of event.
	State is a subtype of state vertex.
	State machine is a subtype of behavior.
	State vertex is a subtype of model element.
	Time event is a subtype of event.
	Time expression is a subtype of expression.
	Transition is a subtype of relationship.
instance of	Instance of is an association between a behavior instance and its behavior, indicating that the behavior instance is a concrete manifestation of the behavior. A behavior instance is an instance of a behavior.
internal transition	Internal transition is a composite aggregation of a state to a collection of transitions. The named transition is an internal transition of the state.
invocation	Invocation is a shared aggregation of an action to an operation. The action is the invocation of the operation.
	Invocation is a shared aggregation of an action to a signal. The action is the invocation of the signal.
occurrence	Occurrence is a shared aggregation of a call event to an operation. A call event is an occurrence of an operation.
	Occurrence is a shared aggregation of a signal event to a signal. A signal event is an occurrence of a signal.
	Occurrence is a shared aggregation of a time event to a time expression. A time event is an occurrence of a time event as manifest in a time expression.

references	References is an association between an action expression and a collection of actions, indicating that the action expression references a given action or actions. An action expression references zero or more actions.
source	Source is a bidirectional association between state vertices and transtions. A state vertex may be the source of multiple transitions, and multiple state vertices may be the source of a single transition.
state variable	State variable is a composite aggregation of a state to a collection of attributes. The attributes are the state variables of the state.
substate	Substate is a composite aggregation of a composite state to a collection of state vertices. The state vertices are the substates of the composite state.
target	Target is a bidirectional association between state vertices and transtions. A state vertex may be the target of multiple transitions, and multiple state vertices may be the target of a single transition.
	Target is a shared association of an action to its target. The instance is the target of the action.
transitions	Transitions is a composite aggregation of a state machine to a collection of transitions. The transitions are the direct transitions of the state machine.
trigger	Trigger is a composite association of a transition to at most one event. The event is the trigger of the transition.
vertices	Vertices is a composite aggregation of a state machine to a collection of state vertices. The state vertices are the immediate state vertices of the state machine.

10.2 BASIC SEMANTICS

Behavior is an abstract subtype of ModelElement. The responsibility of Behavior is to name an observable effect.

BehaviorInstance is a subtype of ModelElement. The responsibility of BehaviorInstance is to specify the concrete manifestation of a Behavior instance. Whereas a Behavior instance provides the specification of an observable effect, a BehaviorInstance instance manifests that Behavior instance in time/space, meaning that the BehaviorInstance instance represents the occurrence of an observable effect that exists in time and space.

Instance of an an association between a BehaviorInstance instance and its Behavior instance. The responsibility of instance of is to specify that the BehaviorInstance instane is a concrete manifestation of the BehaviorInstance. Every Behavior instance may have zero or more BehaviorInstance instances, and every BehaviorInstance instance is the instance of not more than one Behavior instance. In most cases, every BehaviorInstance instance is the instance of exactly one Behavior instance. However, it is possible to specify BehaviorInstance instances that have no Behavior, as is often the case in incomplete and/or evolving models.

StateMachine is a subtype of Behavior. The responsibility of StateMachine is to specify a collection of actions carried out over a sequence of state changes. The name of a StateMachine instance is a Name instance representing the name of the StateMachine; its value may not be a null name. A StateMachine instance defines a name space. None of the parts of a StateMachine instance are visible outside of the StateMachine instance; all of the parts of a StateMachine instance are visible within the StateMachine instance, even if they are nested (in which case their names must be qualified if used outside of their scope).

StateVertex is an abstract subtype of ModelElement. The responsibility of StateVertex is to represent the source or the target of a Transition instance.

Vertices is a composite aggregation of a StateMachine instance to a collection of StateVertex instances. The responsibility of vertices is to specify the immediate StateVertex instances that compose the StateMachine instance. Because each StateMachine instance defines a name space, the names of each StateVertex instance owned by a given StateMachine must be unique.

BooleanExpression is a subtype of Expression. The responsibility of BooleanExpression is to specify an expression that resolves to a Boolean instance.

Transition is a subtype of Relationship. The responsibility of Transition is to specify the passage from one StateVertex instance to another; this passage represents a state change. The name of a Transition instance is a Name instance representing the name of the Transition; its name is typically the null name. The guard attribute of a Transition instance is a BooleanExpression that specifies a condition for the trigger of the Transition instance. The default value of guard is True. The value of guard need not be static, but may involve BooleanExpression instances that are drawn from names that are visible to the Transition instance.

Transitions is a composite aggregation of a StateMachine instance to a collection of Transition instances. The responsibility of Transitions is to specify the immediate and the nested Transitions that compose the StateMachine instance.

Source is a bidirectional association between StateVertex instances and Transition instances. The responsibility of source is to specify the source or sources of a Transition instance. A given StateVertex instance may be the source of zero or more Transition instances, and every Transition instance may have one or more sources. By implication, every Transition instance must have at least one source (there may be no dangling

Transition instances). Furthermore, a given Transition instance may have multiple sources; this is a join, representing a synchronization among all of the sources, each of which must be a concurrent State instance. A Transition instance with multiple sources must have exactly one target.

Target is a bidirectional association between StateVertex instances and Transition instances. The responsibility of target is to specify the target or targets of a Transition instance. A given StateVertex may be the target of zero or more Transition instances, and every Transition instance must have at least one target (there may be no dangling Transition instances). Furthermore, a given Transition instance may have multiple targets; this is a fork, representing a concurrent set of Transition instances leading to concurrent State instances. A Transition instance with multiple targets must have exactly one source; the targets of such a Transition instance must refer to concurrent StateVertex instances.

State is a subtype of StateVertex. The responsibility of State is to specify the condition of an instance at a given moment in time/space. The name of a State instance is a Name instance representing the name of the State; its name must not be the null name. A State instance defines a name space.

ActionState is a subtype of State. The responsibility of an ActionState is to specify a State instance with no substates and exactly one internal Transition instance (on do) with an effect whose ActionExpression instance resolves to an Action instance that invokes exactly one Operation instance.

State variable is a composite aggregation of a State instance to a collection of Attribute instances. The responsibility of state variable is to specify attributes of the state. Every State instance may have zero or more Attribute instances, and every Attribute instance may be a state variable of zero or one State instances. Each state variable is declared in the scope of some enclosing State instance.

Internal transition is composite aggregation of a State instance to a collection of Transition instances. The responsibility of internal transitions is to specify the internal transitions of the State instance. An internal transition represents a Transition instance whose source and target are the same State instance, but with the dynamic semantics that triggering an internal transition does not leave the State instance, nor does it cause that State instance's entry, do, exit transitions to be executed. In fact, as described below, entry, do, and exit are predefined internal transitions. It is possible to define other internal transitions. Internal transitions are just Transition instances, and so may have guards, triggers, and effects.

Pseudostate is a subtype of StateVertex. The responsibility of Pseudostate is to specify all non-state vertices, including initial, final, and history connections. The kind attribute of Pseudostate specifies the kind of the non-state vertex. The default value of kind is initial. An initial Pseudostate instance represents a start state; an initial Pseudostate may not be the target of any Transition instance. There must be exactly one initial Pseudostate instance that is immediately part of a StateMachine instance, as well as exactly one initial

Pseudostate immediately part of every CompositeState instance. Any Transition instance for which an initial Pseudostate instance is a source must not have any trigger (but may have an effect). A final Pseudostate instance represents a final state; a final Pseudostate may not be the source of any Transition instance. A history Pseudostate instance represents a history marker, whose presence affects the dynamic semantics of a State instance as described below; a history Pseudostate instance may be the source of a Transition instance. Final and history Pseudostate instances are optional; each StateMachine and CompositeState instance need not include any Pseudostate instances.

CompositeState is a subtype of State. The responsibility of CompositeState is to specify a state containing one or more substate State instances. The name of a CompositeState instance is a Name instance representing the name of the CompositeState; its name must not be the null name. As a kind of State, a CompositeState instance defines a name space. The isConcurrent attribute of a CompositeState instance is a Boolean that specifies that the substates of the given State instance are considered concurrent. The default value of isConcurrent is False. A concurrent State instance represents an orthogonal state that participates in a flow of control independent of its peer concurrent State instances.

Substate is a composite aggregation of CompositeState instance to a collection of StateVertex instances. The responsibility of substate is to specify the StateVertex instances that are a part of a CompositeState instance. Every CompositeState instance may have one or more substates, and every substate belongs to exactly one CompositeState instance (StateVertex instances that are immediate vertices of a StateMachine instance are not owned by any CompositeState instance).

Event is an abstract subtype of ModelElement. The responsibility of Event is to specify a significant occurrence in time/space. An Event instance never has a Name instance.

Trigger is a composite aggregation of a Transition instance to at most one Event instance. The responsibility of trigger is to specify the Event instance that initiates a state change from the source (or sources) of the Transition instance to its target (or targets). Every Transition instance may have zero or one Event instance, and every Event instance is a part of exactly one Transition instance. Note that it is possible for a Transition instance to have no trigger; such a Transition instance is considered to be an unconditional transition.

BooleanExpression is a subtype of Expression. The responsibility of BooleanExpression is to provide an Expression instance that resolves to a Boolean instance.

TimeExpression is a subtype of Expression. The responsibility of TimeExpression is to provide an Expression instance that resolves to a Time instance. In this context, the Time instance represents an absolute or relative time event that may trigger a Transition instance.

SignalEvent is a subtype of Event. The responsibility of SignalEvent is to specify an Event instance triggered by the invocation (sending) of a Signal instance.

CallEvent is a subtype of Event. The responsibility of CallEvent is to specify an Event instance triggered by the invocation of an Operation instance.

TimeEvent is a subtype of Event. The responsibility of TimeEvent is to specify an Event instance triggered by the passing of time.

Occurrence is a shared aggregation of a SignalEvent, CallEvent, and TimeEvent instance to a Signal, Operation, or TimeExpression instance, respectively. The responsibility of occurrence is to specify the invocation (sending) of a Signal instance, the invocation of an Operation instance, or the time event of a TimeExpression instance that may trigger a Transition instance. Every Signal, Operation, and TimeExpression instance may manifest itself as zero or more Event instances, but every Event instance is the occurrence of exactly one Signal, Operation, or TimeExpression instance.

ActionExpression is a subtype of Expression. The responsibility of ActionExpression is to provide an Expression instance that resolves to the reference of one or more Action instances, where reference denotes a using rather than a defining occurrence of the Action instance. The operation referencedActions returns a list of Action instances referenced by the ActionExpression instance.

References is an association between an ActionExpression and a collection of Action instances, indicating that the ActionExpression instance references a collection of Action instances. The responsibility of references is to establish the relationship between an Action instance and the ActionExpression instances in which it is referenced. Every Action instance may be referenced in zero or more ActionExpression instances, and every ActionExpression instance references zero or more Action instances. References is an implicit relationship, meaning that it is not manifest but rather is derivable from the value of the ActionExpression itself.

Effect is a composite aggregation of a Transition instance to an ActionExpression instance. The responsibility of effect is to specify the effect of a transition. Every Transition instance may have no more than one ActionExpression instance as an effect, and every ActionExpression is the effect of exactly one Transition instance. Note that it is possible for a Transition instance to have no effect.

Action is a subtype of ModelElement. The responsibility of Action is to specify the work carried out as the effect of a state change. An Action instance never has a Name instance.

Invocation is a shared aggregation of an Action instance to a Signal or an Operation instance. The responsibility of invocation is to specify the invocation of a Signal or an Operation instance. Every Signal and Operation instance may manifest itself in zero or more Action instances, but every Action instance is the invocation of exactly one Signal or Operation instance.

Target is shared association of an Action instance to an Instance instance. The responsibility of target is to specify the target of the Signal or Operation instance that is invoked by an Action instance. Every Instance instance may be the target of zero or more Action instances, and every Action instance has no more than one target Instance instance.

Actual argument is a shared association of an Action instance to a collection of Instance instances representing the actual arguments of the Operation instance invoked by an Action instance. Every Instance may be the actual argument of zero or more Action instances, and every Action instance may have zero or more Instance instances as actual arguments. Where an Action instance does have actual arguments and the Action is an invocation of an Operation instance, these Instance instances are ordered, and must match the formal parameters of the corresponding Operation instance in number, order, and by their type. Where an Action instance does have actual arguments and the Action is an invocation of a Signal instance, these Instance instances are ordered, and must match the Attribute instances of the corresponding Signal instance in number, order, and by their type.

A StateMachine instance may be executed; execution represents the dynamic behavior of the StateMachine instance.

A StateMachine instance has run-to-completion processing, meaning that it is not interruptible. A StateMachine instance can react at any given time to exactly one Event instance applied by some external Instance instance. Upon invocation of an Event instance, execution of the StateMachine instance proceeds until it reaches a stable state, whereupon it may wait for another Event instance or react to a new or already queued Event instance. By implication, this means that Event instances that represent the occurrence of a time event do not introduce a time out.

Execution of a StateMachine instance begins with the initial Pseudostate instance that is directly a part of the StateMachine instance. This initial Pseudostate instance must be the sole source of a Transition instance that has no trigger. Execution proceeds with the evaluation of the Transition instance’s effect, if there is one.

It is possible for an initial Pseudostate instance to be the source of multiple Transition instances, each of which has no trigger, but some of which have guards. The choice of which Transition to follow observes the semantics of leaving a State instance as described below. It must always be possible to transition from an initial Pseudostate instance.

Evaluation of an effect involves evaluating every Action instance referenced by the given ActionExpression, in order of its occurrence in the List instance returned by referencedActions. Evaluation of an Action instance involves invoking the associated Signal or Operation instance targeted at the given target Instance instance. For those Action instances that invoke an Operation instance, the Action instance’s actual arguments are used to invoke the associated Operation instance. The evaluation of an effect is not interruptible; all Action instances run to completion. By implication, occurrences of Signal, Operation, or TimeExpression instances that may trigger an Event instance upon the Instance instance that encloses the StateMachine instance are queued.

Immediately upon completion of the evaluation of an effect, execution of a StateMachine instance proceeds to the target StateVertex instances of the Transition instance, according to the following three cases.

First consider the case where a Transition instance has exactly one target, which is an instance of State or ActionState (but not an instance of CompositeState). Execution of the StateMachine instance proceeds with this State instance being entered. If there is an entry internal Transition instance associated with the State instance, its guard attribute is evaluated. If this guard evaluates to True, then the effect of the entry internal Transition instance is evaluated. After completion of the evaluation of this effect, the StateMachine instance is said to be in this State instance. Execution of the StateMachine instance waits until conditions arise that cause a state transition, as described below. At this point, if there are is a do internal transition, the Action instance associated with the do transtion is begun.

Second, consider the case where a Transition instance has exactly one target, which is an instance of CompositeState. Execution of the StateMachine instance proceeds with this CompositeState instance being entered. If there is an entry internal Transition associated with this CompositeState instance, its guard attribute is evaluated. If this guard evaluates to True, then the effect of the entry internal Transition instance is evaluated. After completion of the evaluation of this effect, the StateMachine instance is said to be in this CompositeState instance. At this point, if there is a do transition, the Action instance associated with the do transition is begun. Furthermore, execution of the StateMachine instance proceeds to the initial Pseudostate instance that is directly a part of the CompositeState instance. This initial Pseudostate instance must be the sole source of Transition instance that has no trigger. Execution proceeds with the evaluation of the Transition instance’s effect, if there is one. Thus, the StateMachine instance is said to be in both the CompositeState instance and some substate. Execution of the StateMachine instance waits until conditions arise that cause a state transition, as described below.

Third, consider the case where a Transition instance has more than one target. In such a case, these targets must represent State instances, all but one of which represents a concurrent CompositeState instance. Execution of the StateMachine instance proceeds with all of the concurrent CompositeState instances being entered as above, the difference being that such a transition represents a fork in the flow of control of the StateMachine instance, yielding concurrent flows.

Immediately upon reaching a stable state under any of above three cases, execution of a StateMachine instance proceeds depending upon the presence of any pending Event instances and upon the properties of all Transition instances for which the current State instance is a source, according to the following three cases.

First, consider the case where a State instance is the source of several Transition instances, exactly one of which has no trigger. The guard of this trigger is evaluated exactly once. If this guard evaluates to False, the State instance is not left, and execution of the StateMachine suspends, pending the next Event instance. If the guard evaluates to

True, then execution proceeds with this State instance being left. If there is an exit internal Transition instance associated with the State instance, its guard attribute is evaluated. If this guard evaluates to True, then the effect of the exit internal Transition instance is evaluated. After completion of the evaluation of this effect, execution of the StateMachine instance proceeds with the evaluation of the sole triggerless Transition instance (also known as a spontaneous transition) as above (its guard is not reevaluated, but its effect, if any, is evaluated as above). By implication, this is a transitory State instance; the StateMachine instance is in that State instance for an insignificant amount of time (namely, only during the evaluation of the State instance's entry, do, and exit effect, if any).

Second, consider the case where a State instance is the source of several Transition instances, more than one of which has no trigger. The guards of each of these triggerless Transition instances are evaluated exactly once, in a nondeterministic order not specified by the UML. If all of these guards evaluate to False, the State instance is not left, and execution of the StateMachine instance proceeds as above. If exactly one of these guards evaluates True, the State instance is left and execution of the StateMachine instance proceeds as above, following the one triggerless Transition instance whose guard evaluated True. If more than one of these guards evaluates True, the execution of the StateMachine instance is nondeterministic: execution will proceed as above, following only one of the triggerless Transition instances whose guard evaluated True, the choice of which is implementation dependent.

Third, consider the case where a State instance is the source of several Transition instances, all of which have triggers. Execution of the StateMachine instance proceeds, waiting in the State instance for a pending Event instance. Upon receipt of an Event instance, execution continues with an evaluation of all Transition instances for which that State instance is a source and whose trigger is that Event instance. This yields three possible cases. First, consider the case where there is an Event instance, but no Transition instance is triggered by that Event instance. In such as case, the Event instance is ignored (and in effect discarded) and execution proceeds with the StateMachine in a stable state, specifically, in the same State instance as before the invocation . Second, consider the case where there is an Event instance, and exactly one Transition instance triggered by that Event instance. The guard of this Transition instance is evaluated exactly once. If this guard evaluates to False, the State instance is not left, and the Event instance is ignored (and in effect discarded). If this guard evaluates to True, then the State instance is left and execution of the StateMachine instance proceeds as above, following the one Transition instance triggered by the Event instance. Third, consider the case where there is an Event instance, and more than one Transition instance triggered by that Event instance. The guards of these Transition instances are evaluated in an order not determined by the UML. If exactly one of these guards evaluates True, the State instance is left and execution of the StateMachine instance proceeds as above, following the one Transition instance triggered by the Event instance. If more than one of these guards evaluates True, the State instance is left and execution of the StateMachine instance proceeds as above, following only one of the Transition instances triggered by the Event instance, the choice of which is implementation dependent.

In the event of Event instances that trigger internal Transitions instances (other than the predefined entry, do, and exit internal Transition instances), execution proceeds as above, the difference being that evaluation of an internal Transition instance does not reevaluate the entry, do, and exit internal Transition instances associated with the State instance. The implication of these semantics are that the State instance is not left in the face of an internal Transition instance.

Internal Transition instance should not be confused with self transitions, the latter of which are Transition instance that are not internal transitions but whose source and target are the same State instance. Unlike internal Transition instances, evaluation of a self transition causes reevaluation of the State instance's entry, do, and exit internal Transition instances (if any exist).

The above three cases have further implications, depending upon if the State instance is a substate of a CompositeState instance of if the State instance is a CompositeState with substates.

First, consider the case of the State instance being a substate of a CompositeState. If execution of the StateMachine instance follows a Transition instance from the substate to outside its enclosing CompositeState, then leaving the substate also leaves the CompositeState instance. This means that immediately after the exit internal Transition instance of the substate is evaluated (if one exists), then the exit internal Transition instance of the enclosing CompositeState instance is evaluated (if one exists), and so on, to the outermost CompositeState instance beyond which the Transition instance is targeted.

Second, consider the case of the State instance being a CompositeState with substates. If execution of the StateMachine instance follows a Transition instance from the superstate, then the substates are left first (in order of their nesting, from innermost to outermost), causing evaluation of their exit internal Transition instance (if they exist

There may be conditions whereby both a substate and a superstate are be able to respond to the same Event instance at the same moment in time/space. In such a case, the innermost Transition instance takes priority.

The above semantics address a state change from peer-to-peer State instances, and from inner-to-outer State instances. There are further implications for Transition instances whose source is an outer State instance and whose target is an inner State instance. If execution of a StateMachine instance proceeds with a transition to a substate, then the superstates are entered first (in order of their nesting, from outermost to innermost), causing evaluation of their entry internal Transition instances (if they exist).

Two additional dynamic semantics interact with the above semantics.

First, a final Pseudostate instance may be the target of one or more Transition instances. When the execution of a StateMachine instance reaches a final Pseudostate instance, the

StateMachine instance is said to be terminated, and all future Event instances directed to the StateMachine instance are ignored, and the StateMachine instance manifests no further effects (and represents the destruction on the enclosing Instance instance).

Second, a history Pseudostate instance may be the target of one or more Transition instances. A history Pseudostate instance is never the source of a Transition instance. A CompositeState instance may have at most one immediate history Pseudostate instance. When transitioning to a CompositeState instance which contains a history Pseudostate instance, execution proceeds as above, with the initial Pseudostate instance of the CompositeState instance entered. When transitioning to the history Pseudostate instance of the CompositeState instance, execution does not proceed with the initial Pseudostate instance, but rather, execution proceeds with the innermost substate that was last left upon leaving the CompositeState. Initially, a CompositeState instance has no history, and so the CompositeState instance itself is considered the last left State instance.

10.3 DERIVED SEMANTICS

The semantics of ModelElement are described in section 2.

The semantics of Name, Boolean, Expression, and Time are described in section 4.

The semantics of Instance are described in section 5.

The semantics of Relationship are described in section 6.

The semantics of Signal, Attribute, and Operation are described in section 7. In most cases, a Signal instance has no associated Operation instances.

Note that Signal and Operation appear in this section connected to both Event and Action. Connected to Action, Signal and Operation represent invocations; connected to Event, Signal and Operation represent receipt. Thus, these relationships provide closure: an Event or Operation produced in one StateMachine instance may be consumed in another.

As described in section 5, Signal is a subtype of Class (which in turn is a subtype of Type), and as described in section 6, Type instance may participate in Generalization relationships. Hence, it is common to define hierarchies of Signal instances. These semantics interact with the semantics of triggering a Transition instance as described above. Specifically, triggering a Transition instance via a SignalEvent instance is polymorphic. For example, consider a simple hierarchy with a Signal instance S and its two subtypes S1 and S2. Consider also some Transition instance T whose trigger is specified via a SignalEvent instance. If the trigger of T is S1, then this Transition instance will fire only upon receipt of an instance of S1 (and thus will not fire upon receipt of an instance of S or S2). On the other hand, if the trigger if T is S, then this Transition instance will fire upon receipt of an instance of S or its subtypes, S1 and S2. In this manner, the response to a Signal instance trigger is said to be polymorphic.

Expression is not a subtype of ModelElement, and therefore neither are ActionExpression, BooleanExpression, or TimeExpression.

Behavior/BehaviorInstance are part of the essence/manifestation dichotomy in the UML.

Behavior is a subtype of ModelElement, and therefore Behavior instances may have Stereotype, TaggedValue, Note, and Constraint instances attached (as well as participate in Dependency relationships).

Behavior appears in two other places in the metamodel: in section 5 as part of a Type (and Instance) instance and in section 9 as a part of a Collaboration instance. As part of a Type instance, a Behavior instance states the meaning of the Type instance independent of its realization. In such a context, the target of Action instances in associated StateMachine instances may be absent, in which case the target is assumed to be the Instance instance of the enclosing Type instance. As part of a Collaboration instance, a Behavior instance states the meaning of the society of Type instances that collaborate in the Collaboration instance (and hence are often used to represent the realization of a Type or Operation instance).

There are exactly two subtypes of Behavior, StateMachine and Interaction. The semantics of Interaction are described in section 11. Both kinds of Behavior instances may be used to state the semantics of the Type or Collaboration instance to which the Behavior instances are attached, but in different ways. Whereas a StateMachine instance specifies the semantics of a behavior, an Interaction instance reflects these behavioral semantics. Put another way, an StateMachine instance specifies the potential behavior of the Type or Collaboration instance to which it is attached, whereas an Interaction instance only states a simple path through one behavior under specific circumstances. Thus, a StateMachine instance may yield a large number of corresponding Interaction instances (each showing specific paths through all the potential paths of behavior); furthermore, a set of Interaction instances (each of which states a prototypical behavior) may be combined to specify the StateMachine instances of each of the participants that appear in each of the Interaction instances. In this manner, StateMachine instances and Interaction instances attached to the same Type or Collaboration instance complement one another. Theoretically, only one of these Behavior subtypes is necessary to expression the behavioral semantics of a Type or Collaboration instance, but practically, both are useful, because they give the modeler a choice.

Another important distinction between the semantics of StateMachine and Interaction is the fact that StateMachine instances generally specify the behavior of a single type, whereas Interaction instances typically cut across many such types.

As described in section 11, Behavior instances may include clusters specified by Collaboration instances. These Collaboration instances are typically used to specify “swimlanes” in the corresponding ViewElement instance.

StateMachine instances represent the model behind StateDiagram instances, which are described in section 12; such diagrams represent behavioral diagrams that are organized by state. StateMachine instances whose vertices are only ActionState instances (and not State or CompositeState instances) represent the model behind ActivityDiagram instances, which are described in section 12; such diagrams represent behavioral diagrams that are organized by action.

Because Transition is a subtype of Relationship, the Name instance of each Transition instance must be unique across its sources and targets, as described in section 6 for the semantics of Relationship instance names. Also as described in that section, the semantics of destroying the sources and/or targets of a Transition instance guarantee that there may be no dangling Transition instances.

The Name instance associated with a Transition is typically null. If not null, this Name instance may be used in Constraint instances within the scope of the StateMachine instance to specify time constraints. Furthermore, these names may be used in TimeExpression instances. In both contexts, a Transition name may be used to represent the time of that transition. Specifically, a Name instance N represents the start of the transition (upon the occurrence of the Event instance) and N' represents the end of the effect of the transition (after the completion of all Action instances that form). Using these names permits complex Constraint and TimeExpression instances to be created, involving times relative to an event (for example, `N + 30 microseconds` would represent a time 30 microseconds after N).

The guard BooleanExpression instance associated with a Transition may involve arbitrarily complex expressions. Some of the conditions of such Expression instances may include references to the state of a StateMachine instance. Specifically, the occurrence of the Name instance of a StateVertex instance in a guard BooleanExpression represents a predicate testing if the named StateMachine instance is in that state.

An Event is described as specifying a significant occurrence in time/space and not just in time, because of the interaction of Event semantics with that of Node semantics. As described in section 8, a Node instance represents a physical part upon which Component instances may be deployed. By implication, each Node instance represents an independent processing element in a distributed system. In any such system, relativistic semantics apply, meaning that there is no such thing as a simultaneous event: every observer will see different events. For this reason, Event instances are guaranteed to be unique only within some time/space context, not simply a time context.

As described in section 8, each ActiveClass instance defines a separate event queue. These semantics guarantee that, in the context of an ActiveClass instance, exactly one Event instance will be handled by the associated StateMachine instance at one time. Since State instances are not interruptible, this means that any Event instances that are invoked during the execution of an Action instance are queued. It is possible to write ActionExpression instances that manipulate this queue (for example, to purge the queue, remove an event from the queue, or to query the queue), but such operations are not a part of the core UML, but rather must be modeled as part of the enclosing ActiveClass instance.

The Signal and Operation instances for which an Event instance is the occurrence may be any Signal or Operation instance that is visible to the StateMachine instance, either directly (via the Type instance that owns the Behavior instance) or indirectly (via the Collaboration instance that owns the Behavior instance). Similarly, the Signal and Operation instances which are invoked by an Action instance may be any Signal or Operation instance that is visible to the StateMachine instance, either directly or indirectly. Finally, the Instance instances that are the target or actual arguments of an Action instance may be any Instance instance that is visible to the StateMachine instance, either directly or indirectly. These Instance instances include, but are not limited to state variables, formal parameters of Operation instances visible to the StateMachine instance, an Attributes instances of the Type instance that encloses the StateMachine instance. In this sense, these Instance instances are not fully manifest, but represent prototypical named instances.

A StateMachine instance may include Action instances that send Signal instances to the same Instance instance that encloses the StateMachine instance. This kind of self-referential triggering is allowed; any such Signal instance become an occurrence of an Event instance which may trigger a Transition instance in the same StateMachine instance. Since all such events are queued on the closest ActiveClass instance that encloses the StateMachine instance, this Signal instance will itself be queued.

As described in section 6, Type instance may participate in generalization relationships. Thus, the StateMachine instances owned by such Type instances are themselves inherited. A subtype Type instance inherits the StateMachine instances of its supertype Type instances. The UML specifies no rules about clashes among these inherited StateMachine instances except to say that the semantics of substitutability must still hold, as described in section 6; specifying a formal model here is beyond the scope of practicality. The UML does permit subtype Type instances to add new StateVertex and Transition instances to its inherited StateMachine instances, but again requiring that the semantics of substitutability still hold.

Sending a Signal or Operation instance to an Instance instance that is the instance of a composite type (meaning that the Type instance has composite aggregation relationship to other Type instances) does not implicitly send the Event instance to all of its parts. To achieve this effect, it is necessary that the StateMachine instance associated with the Type instance (for which the Instance instance is an instance of) explicitly state an ActionExpression instance that sends the same trigger instance to its parts.

The run-to-completion semantics of a StateMachine instance interact with the semantics of the predefined do transition. The Action instance associated with a do transition is in fact not interruptable; however, the implementation of an Operation instance associated with the do Action instance may periodically check to see if there are pending events (using the predefined _hasEvent operation), and if there are events, complete its processing so that the event can be handled. Note that these semantics do not specify implicit interruptability - the StateMachine instance runs-to-completion, but the modeler may specify points where the presence of an event is tested.

10.4 STANDARD ELEMENTS

There are no standard stereotypes, tagged values or notes that apply to the metamodel classes described in this diagram.

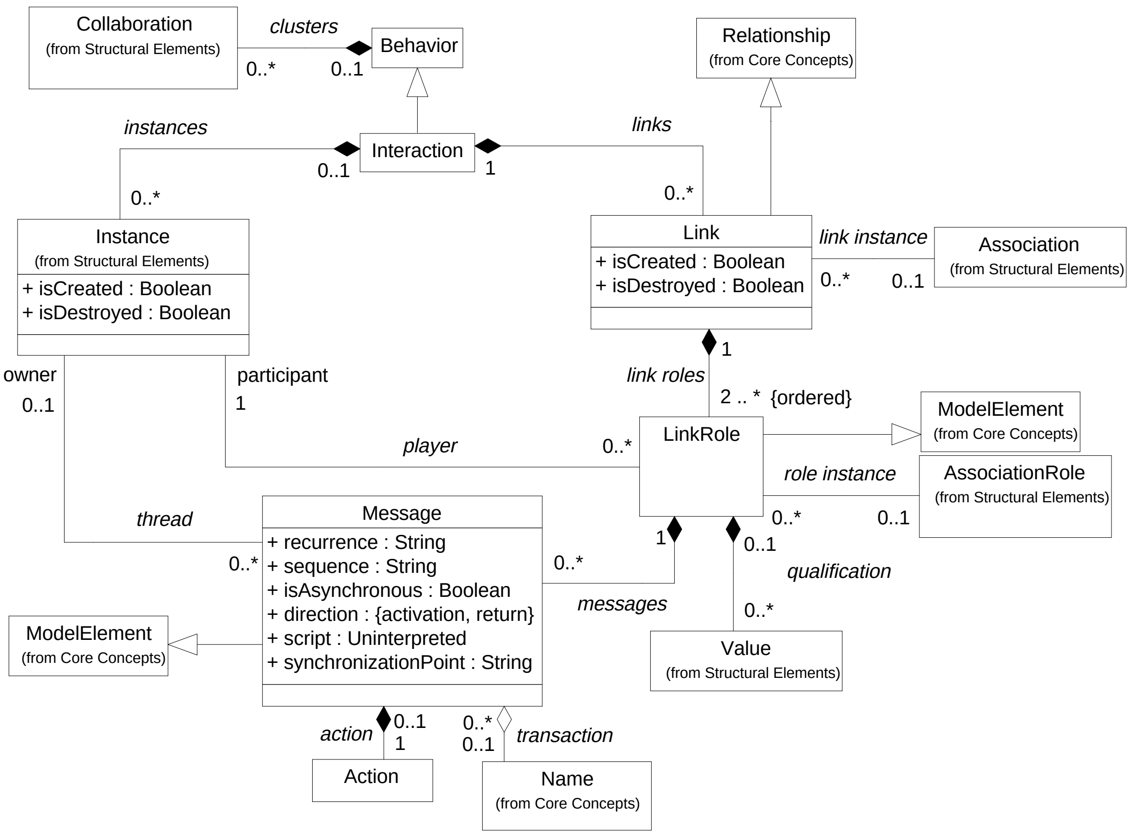
As described in section 10.3, there are three predefined names for Transition instances, namely, `entry`, `do`, and `exit`.

There is one synonym that applies to the metamodel classes described in this diagram:

Synonym	Definition
activation	An activation is the execution of an operation.

11. BEHAVIORAL ELEMENTS: INTERACTIONS

An interaction is a behavior resulting from the colaboration of a collection of instances. An interaction may be viewed from the perspective of time (via sequence diagrams) or of space (via collaboration diagrams). An interaction reflects the behavior of a collaboration.



11.1 DESCRIPTION

This diagram describes the semantics of interactions in the UML, and includes the following metamodel classes:

Action	Described in section 10
Association	Described in section 6
AssociationRole	Described in section 6
Behavior	Described in section 5
Instance	Described in section 5
Interaction	An interaction is a behavior reflected by the collaboration of an interacting society of instances. An interaction reflects the behavior of a collaboration of types.

Link	A link is a relationship among instances across which messages may be sent.
LinkRole	A link role is the face that an instance plays in a link.
Message	A message is the sending of an action.
ModelElement	Described in section 2
Name	Described in section 2
Collaboration	Described in section 9
Relationship	Described in section 3
Value	Described in section 5

This diagram also introduces the following relationships:

action	Action is a composite aggreation of a message to an action. The action is the action of the message.
clusters	Clusters is a composite aggregation of a behavior to a collection of collaborations. The collaborations are clusters of the behavior.
generalization	Interaction is a subtype of behavior.
	Link is a subtype of relationship.
	LinkRole is a subtype of model element.
	Message is a subtype of model element.
instances	Instance is a composite aggregation of an interaction to a collection of instances. The instances are instances that collaborate in the interaction.
link instance	Link instance is an association between a link and an association, indicating that the link is the instance of the association.
link roles	Link roles is a composite aggregation of a link to its link roles. A link has two or more link roles.
links	Links is a composite aggregation of an interaction to a collection of links. The links are the links that connect the collaborators in an interaction.
messages	Messages is a composite aggregation of a link role to a collection of messages. The messages are the messages attached to the link role.

player	Player is an association between an instance and a link role, indicating that the instance plays the given role. The instance is thus a participant in the link.
qualification	Qualification is a composite aggregation of a link role to a collection of values. The vales are the qualification of the link role.
role instance	Role instance is an association between a link role and an associatoin role, indicating that the link role is the instance of the association role.
thread	Thread is an association between an instance and a message, indicating that the instance names the thread in which the given message is invoked. The instance is thus the thread owner.
transaction	Transaction is a shared aggregation of a message to a name. The name is the transaction of the message.

11.2 BASIC SEMANTICS

Interaction is a subtype of Behavior. The responsibility of Interaction is to reflect the behavior of an interacting society of instances. The name of an Interaction instance is a Name instance representing the name of the Interaction; its value may not be a null name. An Interaction instance defines a name space.

Clusters is a composite aggregation of a Behavior instance to a collection of Collaboration instances. The responsibility of clusters is to specify interesting groups of Instance instance and Link instances in the context of a Behavior instance.

Instance is a subtype of ModelElement. The isCreated attribute of an Instance instance is a Boolean that specifies if the Instance instance has just been created. The default value of isCreated is False. The isDestroyed attribute of an Instance instance is a Boolean that specifies if the Instance instance has just been destroyed. The default value of isDestroyed is False. It is possible but rare to have both attribute values True, in which case the Instance instance is first created and then immediately destroyed.

Instances is a composite aggregation of an Interaction instance to a collection of Instance instances. The responsibility of instances is to specify the Instance instances that compose the Interaction instance. Every Interaction instance may have zero or more Instance instances, and every Instance instance may be a part of zero or one Interaction instances. Because each Interaction instance defines a name space, any Instance instances that are instances in the Interaction instance and that have the same name are considered to represent the same instance (but potentially with different values, actions, state value, and roles).

Link is a subtype of Relationship. The responsibility of Link is to represent a relationship among instances across which messages may be sent. The name of a Link instance is a Name instance representing the name of the Link; its name is typically the null name. The isCreated attribute of a Link instance is a Boolean that specifies if the Link instance has just been created. The default value of isCreated is False. The isDestroyed attribute of a Link instance is a Boolean that specifies if the Link instance has just been destroyed. The default value of isDestroyed is False. It is possible but rare to have both attribute values True, in which case the Link instance is first created and then immediately destroyed.

Links is a composite aggregation of an Interaction instance to a collection of Link instances. The responsibility of Links is to specify the Link instances that compose the Interaction instance. Every Interaction may have zero or more Link instances, and every Link instance belongs to exactly one Interaction instance.

Link instance is an association between a Link instance and an Association instance. The responsibility of Link instance is to specify that the Link instance is an instance of the Association instance. Every Link instance is the instance of no more than one Association instance, and every Association instance has zero or more Link instances.

LinkRole is a subtype of ModelElement. The responsibility of LinkRole is to specify the face that an instance plays in a Link instance. The name of a LinkRole instance is a Name instance representing the name of the LinkRole; its name is typically the null name.

Link roles is a composite aggregation of a Link instance to a collection of LinkRole instances. The responsibility of link roles is to specify the roles of the Link instances. For a Link instance that is the instance of an Association instance, the LinkRole instances of the Link instance must match the corresponding AssociationRole instances. Every LinkRole instance is thus the link role of exactly one Link instance, and every Link instance has two or more LinkRole instances, with the same number and corresponding order as the corresponding Association instance. Link roles is an ordered aggregation; by convention, the Instance instance associated with the first LinkRole instance in a Link instance is the implicit sender of any Message attached to that LinkRole instance.

Role instance is an association between a LinkRole instance and an AssociationRole instance. The responsibility of role instance is to specify that the LinkRole instance is an instance of the AssociationRole instance. Every LinkRole instance is the instance of no more than one AssociationRole, and every AssociationRole has zero or more LinkRole instances.

Qualification is a composite aggregation of a LinkRole instance to a collection of Value instances. The responsibility of qualification is to specify the values for the LinkRole instance corresponding to any Attribute instances of the corresponding AssociationRole instance. The Value instances of a LinkRole instance must match the corresponding Attribute instances in order and in type.

Player is an association between an Instance instance and a LinkRole instance. The responsibility of player is to specify that the Instance instance is a player in the LinkRole instance. Every Instance instance may be the player in zero or more LinkRole instances, and every LinkRole instance may be the LinkRole for one Instance instance. The type of the Instance instances must match the type of the corresponding participant of the corresponding AssociationRole instance.

Message is a subtype of ModelElement. The responsibility of Message is to specify the sending of an Action instance. The name of a Message instance is a Name instance representing the name of the Message; its name is typically the null name. The synchronizationPoint attribute of a Message instance is a String that specifies a collection of thread qualified sequence numbers, each of which must match the sequence number of some Message instance in the Interaction instance, the meaning of which is that the Message instance may not be entered during execution until all of the Messages instances referred to in the guard have been reached. The syntax of a synchronizationPoint string is specified according to the following production rules:

```
      synch_string ::= qualified_sequence {‘,’ synch_string}
qualified_sequence ::= [number | thread_name]
                       {‘.’ qualified_sequence}
```

The number represents a sequential order of the sequence within the next higher level of Action instance invocation. Sequences that differ in one integer term are sequentially related at that level of nesting. The thread name represents the name of a concurrent flow of control (which must be some instance of an ActiveClass). The default value of synchronizationPoint is the null String instance.

The sequence attribute of a Message instance is a String instance representing the order of the Message instance in the context of a given thread specified as simple sequence numbers. The syntax of a sequence string is specified according to the following production rules:

```
   sequence_string ::= number {‘.’ sequence_string}
```

The default value of the sequence attribute is the null String instance.

The recurrence attribute of a Message instance is a String instance representing conditional or iterative execution. The syntax of a recurrence string is specified according to the following production rules:

```
 recurrence_string ::= ‘*’ ‘[‘ iteration_clause ‘]’
 recurrence_string ::= ‘[‘ condition_clause ‘]’
```

An iteration clause represents a repeat of the execution of a Message instance at the given nesting depth. A condition clause represents a predicate upon which execution of the Message instance is contingent. Both the iteration clause and the condition clause are meant to be expressed in pseudocode or an actual programming language; their precise syntax is outside the scope of the UML.

The default value of the recurrence attribute is the null String value.

The isAsynchronous attribute of a Message instance is a Boolean instance representing if the Message instance is sent synchronously or not. The default value of isAsynchronous is False, meaning that the Instance instance that invoked the Message instance suspends until the action of the Message instance is complete. A value of True means that the Instance instance that invoked the Message instance does not wait for the action of the Message instance to complete.

The direction attribute of a Message instance is an enumeration representing if the Message instance is an activation or a return. The default value of direction is activation.

The script attribute of a Message instance is an Uninterpreted instance, the purpose of which is to represent pseudocode describing the meaning of the Message instance in the context of the Interaction instance.

Messages is a composite aggregation of a LinkRole instance to a collection of Message instances. The responsibility of messages is to specify the Message instances attached to the LinkRole instance. Ever LinkRole instance may have zero or more Message instances, and every Message instance is the message of exactly one LinkRole instance.

Action is a composite aggregation of a Message instance to an Action instance. The responsibility of action is to specify the Action instance of a Message instance. Every Message instance has exactly one Action instance, and every Action instance is the action of zero or one Message instances.

Transaction is a shared aggregation of a Message instance to a Name instance. The responsibility of transaction is to group a collection of Message instance as a transaction, meaning that the group is treated as an behavior that can be rolled back.. Every Message instance may have no more than one Name instance representing a transaction, and every Name instance representing a transaction may relate to zero or more Message instances.

Thread is an association between an Instance and a Message instance. The responsibility of thread is to specify the flow of control under which the Message instance is invoked. The Instance instance must be an instance of an ActiveClass instance; the name of the instance is the name of the thread. Every Instance instance may be the thread for zero or more Message instances, and every Message instance may be associated with no more than one thread. Often, the thread wherein a Message instance executes is not made explicit.

An Interaction instance reflects the execution of a sequence of Message instances; execution represents the dynamic behavior of a society of Instance instances.

Execution of an Interaction instance begins with all Message instances whose sequence number is 1 for each independent thread. Activation of a Message instance is the evaluation of an Action instance. Execution proceeds to the next sequentially numbered Mesage instance, tempered by any guards, iteration clauses, and condition clauses which may suspend, repeat, or branch a flow of control, respectively. Starting a nested sequence number represents a nested activation.

11.3 DERIVED SEMANTICS

The semantics of ModelElement are described in section 2.

The semantics of Name are described in section 4.

The semantics of Instance and Value are described in section 5.

The semantics of Relationship, Association, and AssociationRole are described in section 6.

The semantics of Collaborations are described in section 9. In this context, Collaboration instances that serve as the clusters of a Behavior instance are typically used to specify “swimlanes” in the corresponding ViewElement instance.

The semantics of Behavior are described in sections 9 and 10. As described in section 10, Behavior/BehaviorInstance are part of the essence/manifestation dichotomy of the UML.

The semantics of Action are described in section 10.

Link, LinkRole, and Message are all subtypes of ModelElement, and therefore instances of each may have Stereotype, TaggedValue, Note, and Constraint instances attached (as well as participate in Dependency relationships).

As described in section 10, StateMachine and Interaction are the two subtypes of Behavior. Interaction instances reflect the behavioral semantics of a Type or Collaboration instance. As such, an Interaction instance states a prototypical behavior. From collections of Interaction instances, it is possible to compose a partial or full StateMachine instance for each of the Instance instances that participate in the Interaction instances.

Interaction instances represent the model behind InteractionDiagram and CollaborationDiagram instances, which are described in section 12; such diagrams represent behavioral diagrams that are organized by time and by space, respectively.

Parts of Interaction instances may be related to other Interaction instances. For example, a Link instance in an Interaction instance may be the instance of an Association instance that is the reification of a Collaboration instance. Thus, this Link instance may expand to another Interaction instance. Similarly, the Operation instance invoked by the Action instance of a Message instance may expand to another Interaction instance, since as described in section 9, an Operation instance may be represented by a Collaboration instance, which in turn may have Behavior instances.

As described in section 9, Collaboration instances share Instance and Relationship instances. This means that it is possible, within the same Interaction instance, to group these Element instances in arbitrary and overlapping ways.

The name instance associated with a Message is typically null. If not null, this Name instance may be used in Constraint instances within the scope of the Interaction instance to specify time constraints. In this context, a Message name N represents the start of the message and N' represents the end of the evaluation of the Message instance.

The isAsynchronous attribute of Message distinguishes between synchronous and asynchronous messages. Other forms of synchronization may be specified using appropriate TaggedValue instance, none of which are defined as standard elements of the UML

Instance instances with the same Name instance in a given Interaction instance represent the same instance. Dependencies among such same-named instances may be shown, most often using the becomes and copy stereotypes as described in section 5. Such instances may be rendered with potentially with different values, actions, state value, and roles.

Type/Instance, Association/Link, AssociationRole/LinkRole, Attribute/Value, Signal/Message, and Operation/Message each form an essence/manifestation pair.

11.4 STANDARD ELEMENTS

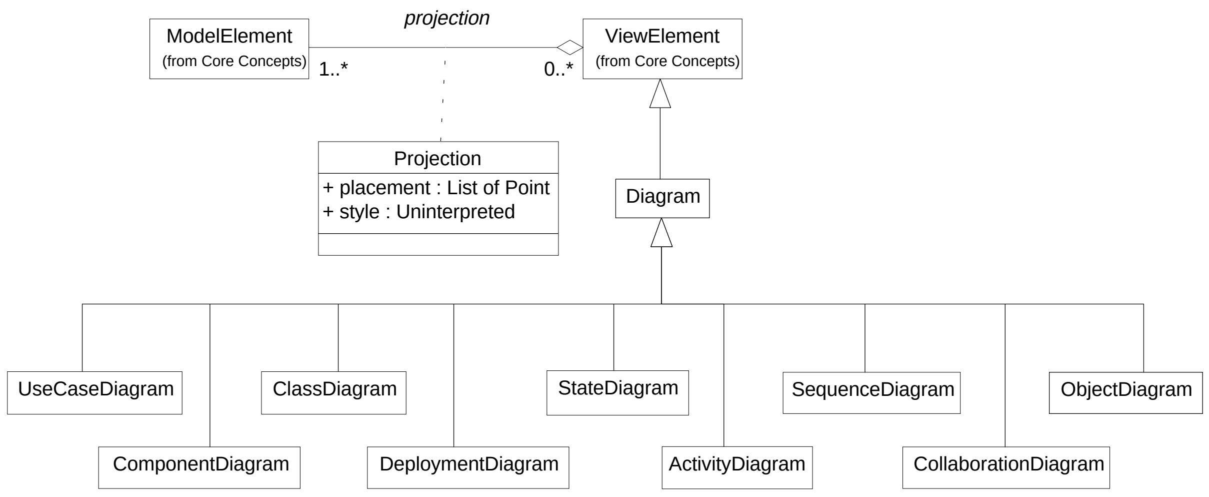
There are seven standard constraints that apply to the metamodel classes described in this diagram:

Name	Applies to	Semantics
association	LinkRole	Association is a constraint applied to a link role, specifying that the corresponding instance is visible via association.
broadcast	Message	Broadcast is a constraint applied to a message, specifying that it is not invoked in any specific order.
global	LinkRole	Global is a constraint applied to a link role, specifying that the corresponding instance is visible because it is in a global scope.
local	LinkRole	Local is a constraint applied to a link role, specifying that the corresponding instance is visible because it is a local variable of an operation.

parameter	LinkRole	Parameter is a constraint applied to a link role, specifying that the corresponding instance is visible because it is a parameter if an operation.
self	LinkRole	Self is a constraint applied to a link role, specifying that the corresponding instance is visible because it is the dispatcher of a message or a part of the dispatcher.
vote	Message	Vote is a constraint applied to a collection of return messages, specifying that the return value is selected by a majority vote of all the return values in the collection..

12. VIEW ELEMENTS: VIEW ELEMENTS

A view element is a textual and graphical projection of a collection of model elements. The UML predefines a number of such graphical projections as common diagrams.



12.1 DESCRIPTION

This diagram describes the view elements of the UML, and includes the following metamodel classes:

ActivityDiagram	An activity diagram encompasses states and their relationships, organized by actions, and is used to specify the behavior of an operation.
ClassDiagram	A class diagram encompasses types, classes, and their relationships.
CollaborationDiagram	A collaboration diagram encompasses instances and their relationships (including messages), organized by space.
ComponentDiagram	A component diagram encompasses components and their relationships.

DeploymentDiagram	A deployment diagram encompasses components, nodes, and their relationships.
Diagram	A diagram is a graphical projection of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).
ModelElement	Described in section 2
ObjectDiagram	An object diagram encompasses instances and their relationships.
Projection	Described in section 2
SequenceDiagram	A sequence diagram encompasses instances and their relationships (including messages), organized by time.
StateDiagram	A state diagram encompasses states and their relationships, organized by state, and is used to specify the overall behavior of a type.
UseCaseDiagram	A use case diagram encompasses actors, use cases, and their relationships.
ViewElement	Described in section 2

This diagram also introduces the following relationships:

generalization	Activity diagram is a subype of diagram.
	Class diagram is a subtype of diagram.
	Collaboration diagram is a subtype of diagram.
	Component diagram is a subtype of diagram.
	Deployment diagram is a subtype of diagram.
	Diagram is a subtype of view element.
	Object diagram is a subtype of diagram.
	Sequence diagram is a subtype of diagram.
	State diagram is a subtype of diagram.
	Use case diagram is a subtype of diagram.
projection	Described in section 2

12.2 BASIC SEMANTICS

The responsibility of ViewElement is to provide a textual and graphical projection of a collection of ModelElement instances. Projection in this context means that the ViewElement instance represents a human readable notation for the corresponding

ModelElement instances. The human readable notation for each ModelElement instance is not defined in the UML metamodel, but rather is defined in the UML notation summary.

Projection is a shared aggregation of a ViewElement instance to a collection of ModelElement instances. For each projection of a ViewElement instance over a ModelElement instance, there is a Projection instance. The responsibility of Projection is to specify the placement and style of a ModelElement instance in the context of a ViewElement instance. The placement attribute of Projection provides a list of Point instances whose value specifies the placement of the projection of the ModelElement instance in the ViewElement instance. For all ModelElement instances that are projected as icons, symbols, and strings, this placement can generally be specified in a single Point instance. For all ModelElement instances that are projected as paths, this placement can generally be specified in a list consisting of two or more Point instances. The style attribute of Projection provides a value of an Uninterpreted instance whose purpose is to specify the style of the projection, including but not limited to indicating the shape, color, typeface, elision, and sound associated with the corresponding ModelElement instance. The implication of this attribute is that the same ModelElement instance may be projected in two or more ViewElement instances, but appear to the reader with different visual cues. The default placement of the projection of a ModelElement instance is the origin point and the default style is the null uninterpreted value.

Diagram is an abstract subtype of ViewElement. The responsibility of Diagram is to provide a graphical projection of a collection of ModelElement instances, most often rendered as a connected graph of arcs and vertices. Implementations are free to introduce new subtypes of ViewElement, although the most common projections are addressed by Diagram.

The ModelElement instances projected in a given Diagram instance may be drawn from any part of a System instance. In other words, the contents of a given Diagram instance may transcend any and all boundaries of Model instance packaging and visibility. These semantics have three important implications. First, Diagram instances may include projections of ModelElement instances from distant Package instances, even if the contents of each of those Package instances are not visible to one another (but they are always visible to the modeler). Second, the projection of a given ModelElement may be elided, meaning that the properties of that ModelElement may be selectively projected, with some parts shown and other parts suppressed. For example, a Diagram instance may have a projection of a Class instance, but with only the associated Stereotype instance, Name instance, and public Member instances rendered. Third, the projection of a given ModelElement instance may be expanded, meaning that derived properties of that ModelElement instance may be selectively projected, showing certain parts reached by walking the metamodel relationships from that first instance. For example, a Diagram instance may be a projection of a Class instance, but with some of the Member instances inherited from all of its supertypes also shown.

A Diagram instance may project other than ModelElement instances. For example, n-ary Association instances, forked and joined Transition instances, and incomplete Generalization instance models require visual elements that can be derived from a Model instance but are not themselves manifest in a Model instance.

In practice, there are a small number of common groupings of ModelElement instances within a Diagram instance. Because these groupings are generally useful and encourage the recommended process associated with the UML, they are predefined in the UML metamodel as the eight subtypes of Diagram described above. This is an incomplete subtyping, meaning that the UML does not prohibit the creation of other kinds of diagrams. The contents of each of these diagrams is described above. Note that these are general groupings: for example, a StateDiagram instance is defined to encompass states and their relationships, but it is not unusual for such a diagram to include instances and types as well.

12.3 Derived Semantics

The semantics of List, Point, and Uninterpreted are described in section 3.

The semantics of ModelElement are described in section 2.

ViewElement is a subtype of Element, and therefore all ViewElement (and by implication, Diagram) instances may have Name, Stereotype, TaggedValue, Note, and Constraint instances attached. The only meaningful kind of Dependency instances that may involve ViewElement instances are trace Dependency instances. ViewElement is not a subtype of ModelElement, and therefore a ViewElement instance may not provide a projection of ViewElement instances.

ObjectDiagram is essentially a degenerate kind of CollaborationDiagram, wherein only the structural aspects of instances are shown. ComponentDiagram is essentially a degnerate kind of DeploymentDiagram, wherein only components (without their deployment to nodes) are shown.

Additional kinds of diagrams can be specified as a stereotyped Diagram instance or as a stereotype of one of the predefined kinds of Diagram subtypes.

12.4 Standard Elements

There are two synonyms that apply to the metamodel classes described in this diagram:

Synonym	Definition
interaction diagram	An interaction diagram is a sequence diagram or a collaboration diagram.
state chart	A state chart is a state diagram.

13. STANDARD ELEMENTS

13.1 Description

Stereotypes, tagged values, and constraints are the mechanisms of extensibility in the UML. Stereotypes extend the classes in the metamodel, tagged values extend the attributes of classes in the metamodel, and constraints extend the semantics of the metamodel. Certain stereotypes, tagged values, and constraints are predefined in the UML; others may be user defined. This section summarizes all of the predefined stereotypes, tagged values, and constraints of the UML, which collectively are called the standard elements of the UML.

13.2 Basic Semantics

The semantics of stereotypes, tagged values, and constraints are described in section 3.

All of the predefined stereotypes, tagged values, and constraints of the UML either have strong semantics that interact with other elements of the UML metamodel or are important properties that transcend any specific development process. Others stereotypes, tagged values, and constraints may be layered on top of the UML to provide process-specific tailorings.

13.3 Derived Semantics

The mechanisms of extensibility are a fundamental part of the UML. Therefore, user defined stereotypes, tagged values, and constraints may be a part of any well-formed model. However, any semantics attached to these user defined elements are guaranteed only for the System instance in which these elements are defined. The UML does not nor cannot require or rely upon the semantics of any user defined or extended stereotype, tagged value, or constraint.

13.4 Standard Elements

The following 35 stereotypes are predefined in the UML:

Name	Applies to	Defined in
actor	Type	5.4
application	Component	8.4
becomes	Dependency	5.4
bind	Dependency	7.4
	Collaboration	9.4

call	Dependency	7.4
constraint	Note	3.4
copy	Dependency	5.4
derived	Dependency	2.4
document	Component	8.4
enumeration	PrimitiveType	5.4
extends	Generalization	6.4
facade	Package	2.4
file	Component	8.4
friend	Dependency	2.4
import	Dependency	2.4
instance	Dependency	5.4
interface	Type	5.4
library	Component	8.4
metaclass	Dependency	7.4
	Type	7.4
page	Component	8.4
powertype	Dependency	6.4
	Type	6.4
process	ActiveClass	8.4
refinement	Dependency	5.4
requirement	Note	3.4
role	Dependency	6.4
send	Dependency	7.4
signal	Class	5.4
stub	Package	2.4
subclass	Generalization	6.4
subtype	Generalization	6.4
table	Component	8.4
thread	ActiveClass	8.4
trace	Dependency	2.4
uses	Dependency	6.4

utility	Type	7.4

The following 10 tagged values are predefined in the UML:

Name	Value	Applies to	Defined in
documentation	String	Element	2.4
invariant	Uninterpreted	Type	7.4
location	Component	ModelElement	7.4
	Node	Component	7.4
persistence	Enumeration	Type	5.4
		Instance	5.4
		Attribute	5.4
postcondition	Uninterpreted	Operation	7.4
precondition	Uninterpreted	Operation	7.4
responsibility	String	Type	5.4
semantics	Uninterpreted	Type	7.4
		Operation	7.4
space semantics	Uninterpreted	Type	7.4
		Operation	7.4
time semantics	Uninterpreted	Type	7.4
		Operation	7.4

The following 14 constraints are predefined in the UML:

Name	Applies to	Defined in
association	LinkRole	11.4
broadcast	Message	11.4
complete	Generalization	6.4
disjoint	Generalization	6.4
global	LinkRole	11.4
implicit	Association	6.4
incomplete	Generalization	6.4
local	LinkRole	11.4
or	Association	6.4

ordered	AssociationRole	6.4
overlapping	Generalization	6.4
parameter	LinkRole	11.4
self	LinkRole	11.4
vote	Message	11.4

Index