**Date:** October 2020

# Kernel Modeling Language (KerML)

*Version 1.0*
*Release 2020-09*

**Submitted in partial response to Systems Modeling Language (SysML®) v2 RFP (ad/2017-12-02) by:**


88Solutions Corporation

Dassault Systèmes

GfSE e.V.

IBM

INCOSE

InterCax LLC

Lockheed Martin Corporation

MITRE

Model Driven Solutions, Inc.

PTC

Simula Research Laboratory AS

Thematix

# Table of Contents

# List of Tables

# List of Figures

# 0 Submission Introduction

## 0.1 Submission Overview

This document is the first of two documents submitted in response to the Systems Modeling Language (SysML®) v2 Request for Proposals (RFP) (ad/2017-11-04). This document defines a *Kernel Modeling Language (KerML)* that provides a syntactic and semantic foundation for creating application specific modeling languages. The second document specifies the *Systems Modeling Language (SysML), version 2.0*, built on this foundation.

Even though both documents are being submitted together to fulfill the requirements of the RFP, the present document for KerML is proposed as a separate specification from SysML v2. KerML provides a common basis for creation of new modeling languages (or evolution of existing modeling languages). It moves beyond the syntactic interoperability offered by MOF to the possibility of diverse modeling languages that tailored to specific application while maintaining fundamental semantic interoperability.

**Release note.** The present document is an update to the initial submission document submitted to OMG in August 2020.

## 0.2 Submission Submitters

The following OMG member organizations are jointly submitting this proposed specification:

- 88Solutions Corporation
- Dassault Systèmes
- GfSE e.V.
- IBM
- INCOSE
- InterCax LLC
- Lockheed Martin Corporation
- MITRE
- Model Driven Solutions, Inc.
- PTC
- Simula Research Laboratory AS
- Thematix

The submitters also thankfully acknowledge the support of over 60 other organizations that participated in the SysML v2 Submission Team (SST).

## 0.3 Submission - Issues to be discussed

*6.7.1 Proposals shall describe a proof of concept implementation that can successfully execute the test cases that are required in 6.5.4.*

The SST is developing a pilot implementation of the full KerML abstract syntax and textual concrete syntax. There have been four quarterly public releases of this pilot implementation so far, the last being the 2020-06 version released at the beginning of July 2020. However, since the conformance test suite has not been developed as of the time of this initial submission, it is not possible to formally demonstrate the conformance of the implementation to the proposed specification. Nevertheless, with few exceptions, this proposed specification describes the language as it has been implemented. For those specific areas in which the pilot implementation as of the 2020-06 release is known to not fully conform to the initial submission of the KerML specification, the deviations are identified in "implementation notes" in this document. The SST is currently planning on releasing the 2020-09 version of the pilot implementation as open source, at which time it is intended that the implementation be fully conformant with the initial submission of this specification.

*6.7.2 Proposals shall provide a requirements traceability matrix that demonstrates how each requirement in the RFP is satisfied. It is recognized that the requirements will be evaluated in more detail as part of the submission process. Rationale should be included in the matrix to support any proposed changes to these requirements.*

See subclause 0.4 in the proposed *Systems Modeling Language (SysML), Version 2.0* specification document submitted along with the present document.

*6.7.3 Proposals shall include a description of how OMG technologies are leveraged and what proposed changes to these technologies are needed to support the specification.*

As required in the SysML v2 RFP, the abstract syntax for KerML is defined as a model that is consistent with the OMG Meta Object Facility [MOF] as extended with MOF Support for Semantic Structures [SMOF] (see 7.1.4). This also allows KerML models represented in the KerML abstract syntax to be interchange using OMG XML Metadata Interchange [XMI].

The OMG MOF standard has been used to define many OMG-standardized modeling languages, and the KerML language definition is also built on it. However, MOF and XMI only standardize the means for specifying the abstract syntax of a modeling language and interchanging models so specified. Even SMOF provides only limited additional support for the syntactic structures required for so-called "semantic" languages.

The goal of KerML is to go beyond this and to become a new OMG standard providing application-independent syntax *and semantics* for creating more specific modeling languages (as described further in Clause 1). This will allow not only syntactic interchange between modeling tools, but also semantic interoperability. The KerML specification is being submitted as part of the SysML v2 submission, because the SST has built SysML v2 on KerML in exactly this way.

## 0.4 Language Requirement Tables

See subclause 0.4 of the proposed *Systems Modeling Language (SysML), Version 2.0* specification document submitted along with the present document.

# 1 Scope

The Kernel Modeling Language (KerML) provides an application-independent syntax and semantics for creating more specific modeling languages. *Modeling languages* are for expressing *models* of some (real or virtual) system of interest. Subclause 6.1 outlines the relationship of modeling languages, models, and modeled systems.

The KerML *metamodel* includes concrete and abstract syntax for KerML (see Clause 7). The concrete syntax provides a notation for expressing system models, while the abstract syntax derived from it is given semantics. Application specific modeling languages can be build on KerML by extending the abstract syntax, specializing its semantics, with concrete syntaxes similar to or entirely different from KerML's.

The specification also includes *model libraries* expressed in KerML concrete syntax (see Clause 8). These capture typical semantic patterns (such as asynchronous transfers and state-based behavior) that can be reused by languages built on KerML. Specialized modeling languages can provide additional syntax for these libraries, tailored to their applications, with semantics based largely or entirely on  the KerML libraries.

The circularity of KerML model libraries expressed in KerML itself is broken by the mathematical semantics of a small *core* subset of the language (see 7.3). The parts of the metamodel built on the core have its mathematical semantics by specialization. This means the KerML libraries have this grounding, providing a consistent basis for mathematical reasoning about models based on these libraries.

# 2 Conformance

This specification defines the Kernel Modeling Language (KerML), a language used to construct *models* of (real or virtual, planned or imagined) things. The specification includes this document and the content of the machine-readable files listed on the cover page. If there are any conflicts between this document and the machine-readable files, the machine-readable files take precedence.

A *KerML model* shall conform to this specification only if it can be represented according to the syntactic requirements specified in Clause 7. The model may be represented in a form consistent with the requirements for the KerML concrete syntax, in which case it can be parsed (as specified in Clause 7) into an abstract syntax form, or may be represented only in an abstract syntax form (see also 7.1.3 and 7.1.4).

A *KerML modeling tool* is a software applications that creates, manages, analyzes, visualizes, executes or performs other services on KerML models. A tool can conform to this specification in one or more of the following ways.

1. *Abstract Syntax Conformance.* A tool demonstrating Abstract Syntax Conformance provides a user interface and/or API that enables instances of KerML abstract syntax metaclasses to be created, read, updated, and deleted. The tool must also provide a way to validate the well-formedness of models that corresponds to the constraints defined in the KerML metamodel. A well-formed model represented according to the abstract syntax is syntactically conformant to KerML as defined above. (See Clause 7.)

2. *Concrete Syntax Conformance.* A tool demonstrating Concrete Syntax Conformance provides a user interface and/or API that enables instances of KerML concrete syntax notation to be created, read, updated, and deleted. Note that a conforming tool may also provide the ability to create, read, update and delete additional notational elements that are not defined in KerML. Concrete Syntax Conformance implies Abstract Syntax Conformance, in that creating models in the concrete syntax acts as a user interface for the abstract syntax. However, a tool demonstrating Concrete Syntax Conformance need not represent a model internally in exactly the form modeled for the abstract syntax in this specification. (See Clause 7.)

3. *Semantic Conformance.* A tool demonstrating Semantic Conformance provides a demonstrable way to interpret a syntactically conformant model (as defined above) according to the KerML semantics, e.g., via model execution, simulation, or reasoning, when and only when such interpretations are possible. Semantic Conformance implies Abstract Syntax Conformance, in that the semantics for KerML are only defined on models represented in the abstract syntax. (See Clause 7 and Clause 8. See also 6.1 for further discussion of the interpretation of models and their syntactic and semantic conformance.)

4. *Model Interchange Conformance.* A tool demonstrating model interchange conformance can import and/or export syntactically conformant KerML models (as defined above) in one or more of the formats specified in Clause 9.

Every conformant KerML modeling tool shall demonstrate at least Abstract Syntax Conformance and Model Interchange Conformance. In addition, such a tool may demonstrate Concrete Syntax Conformance and/or Semantic Conformance, both of which are dependent on Abstract Syntax Conformance.

For a tool to demonstrate any of the above forms of conformance, it is sufficient that the tool pass the relevant tests from the Conformance Test Suite specified in Annex A.

# 3 Normative References

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

[Alf] *Action Language for Foundational UML (Alf),* Version 1.1
https://www.omg.org/spec/ALF/1.1

[DOL] *Distributed Ontology, Model, and Specification Language,* Version 1.0
https://www.omg.org/spec/DOL/1.0

[MOF] *Meta Object Facility,* Version 2.5.1
https://www.omg.org/spec/MOF/2.5.1

[OCL] *Object Constraint Language,* Version 2.4
https://www.omg.org/spec/OCL/2.4

[SMOF] *MOF Support for Semantic Structures,* Version 1.0
https://www.omg.org/spec/SMOF/1.0

[SysAPI] *Systems Modeling Application Programming Interface (API) and Services*
(as submitted contemporaneously with this proposed KerML specification)

[UUID] *A Universally Unique IDentifier (UUID) URN Namespace*
https://tools.ietf.org/html/rfc4122

[XMI] *XML Metadata Interchange,* Version 2.5.1
https://www.omg.org/spec/XMI/2.5.1

# 4 Terms and Definitions

There are no terms and definitions specific to this specification.

# 5 Symbols

There are no symbols defined in this specification.

# 6 Introduction

## 6.1 Language Architecture

Developing systems involves at least two kinds of specifications, one giving the intended effects of a system (requirements), and another determining how it will bring about those effects (design). Many designs might be developed and evaluated against the same requirements. A third kind of specification describes test procedures that check whether requirements are met by real or virtual systems built and operated according to some design. These cover common situations of system operation, but usually cannot cover all of them.

In the terms above, this specification serves as requirements for the KerML language, while implementations of it are analogous to designs. The specification includes a *metamodel* that defines how models are structured (syntax) and *model libraries* that specify how real or virtual things are constructed or operated according to those models (semantics). This language architecture enables two kinds of automatic testing of implementation conformance to this specification, as illustrated in Fig. 1 (also see Clause 2).

1. *Syntactic conformance* is short for models conforming to metamodels. The example model in the middle left of Fig. 1 is expressed in the syntax of KerML at the top (concrete and well as abstract syntax, see 7.1.1), as shown by the upward arrow in the middle. KerML syntax is expressed in the Meta-Object Facility [MOF], enabling the model to be automatically checked for conformance to it.
2. *Semantic conformance* is short for real or virtual things conforming to models in the way they are constructed and during their operation (applies only to syntactically conformant models). Models expressed in KerML reuse elements of the KerML model libraries to give them semantics, as shown my the horizontal block arrow in Fig. 1. These libraries give conditions for conformant things, as built or operated, which are augmented in the model as appropriate.

Semantic conformance helps people interpret models in the same way, because the models extend libraries expressed in a small (core) subset of the same language (as shown in the figure by the arrow at the top right). This subset is the first part of the language that engineers and tool builders learn, enabling them to inspect the libraries to understand the real or virtual effects of things built and operated according to models extending the libraries. More uniform model interpretation improves communication between everyone involved in modeling, including modelers and tool builders.

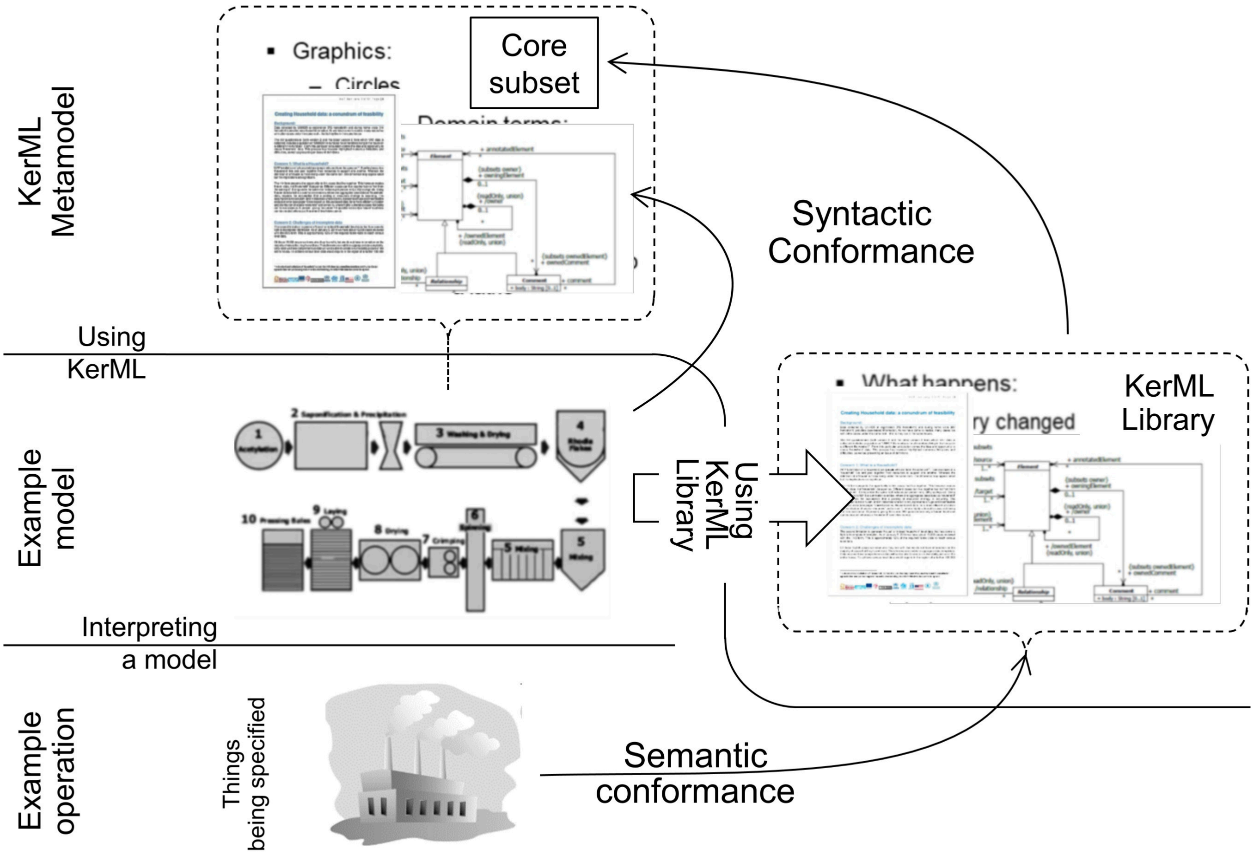

**Figure 1. Syntactic and Semantic Conformance**

## 6.2 Document Conventions

The following stylistic conventions apply to text about the Clause 7 (Metamodel):

1. When names of metaclasses from the KerML abstract syntax are used as common nouns ("an Element", "multiple FeatureTypings") they refer to instances of the metaclass (in models). For example, "Elements can own other Elements" refers to instances of the metaclass Element that reside in models. When italicized or modified by "metaclass" they are proper nouns referring to a metamodel element, rather than its instances. For example, "The Element metaclass and *Relationship* are contained in the Root package." refers to metaclasses.
2. Names of properties of metaclasses appear in "`code`" font. When in regular (non-italic) style, they are common nouns referring the values of the properties (in models), pluralized where necessary, e.g., "the `ownedRelatedElements` of a Relationship". When italicized they are proper nouns referring to a metaelement, rather than it values.

The following stylistic conventions apply to text about the Clause 8 (Model Library):

1. Convention 1 above applies to Types in the KerML library, where the instances are (real of virtual) things of that Type
2. Convention 2 above applies to KerML Features in the KerML Library, where the values are (real of virtual) things.

See 7.3 about instances (interpretations) of KerML Types and Features.

The following conventions apply to the Concrete Syntax subclauses in Clause 7 for the KerML textual notation:

1. Textual notation appears in "`code`" font.
2. Keywords are appear in **`boldface`**, ("Features are declared using the **`feature`** keyword.")

3. Symbols (such as + and `:>>`) and short segments of textual notation (but longer than an individual name) may be written in-line in body text (without being code or bold).
4. Longer samples of textual notation are written in separate paragraphs, indented relative to body paragraphs.

The grammar of the textual Concrete Syntax and its mapping to the Abstract Syntax is expressed in a specialized *Extended Backus-Naur Form* (EBNF) notation described in 7.1.3.

Core mathematical semantics is expressed in the usual notation for first order logic, except:

1. Quantifiers can include the set of which each the variable is a member, rather than leaving this to the body of the statement ($\forall t_g \in V_T$ ... is short for $\forall t_g \; t_g \in V_T \Rightarrow$ ...). The same set can be given once for multiple variables ($\forall t_g, t_s \in V_T$ ... is short for $\forall t_g, t_s \; t_g \in V_T \wedge t_s \in V_T \Rightarrow$ ...).
2. Dots (.) appearing between property or feature names have the same meaning as in OCL, including implicit collections [OCL].

Mathematical terms used in the specification are defined in 7.3.1.2.

**Submission Note.** Paragraphs marked like this one are not part of the proposed specification. They are material that was not included at the time of the submission, or changes that are expected before the next revision. These notes will be removed in revised submissions as they are addressed.

**Implementation Note.** Paragraph marked like this one are not part of the proposed specification either. They identify areas which the proof-of-concept pilot implementation (being developed by the submission team) is not fully consistent with the proposed specification.

**Release Note.** Paragraphs marked like this one provide additional information on the status of updates to this specification document in releases since the initial submission.

## 6.3 Document Organization

The remainder of this document is organized into three major clauses.

- Clause 7 specifies the Metamodel that defines the KerML language. The first subclause of this clause is an overview, with each following subclause describing succeeding layers of the metamodel. The subclaus for each metamodel layer is then divided into an overview and a description of the metamodel elements for each package in the layer (see also 7.1.4). Each package subclause describes the concrete syntax, abstract syntax and semantics of the elements in the package (except that the elements in the Root layer have not model-level semantics).
- Clause 8 specifies the Kernel Model Library, which is a set of KerML models used to provide Kernel-layer semantics to user models. The first subclause of this clause is an overview, with each following subclause describing the elements in a single package in the Model Library, referred to as a *library model.*
- Clause 9 describes each of the formats that can be used to provide standard interchange of KerML models between modeling tools.

In addition, Annex A defines the suite of conformance tests that may be used to demonstrate the conformance of a modeling tool to this specification (see also Clause 2).

## 6.4 Acknowledgements

This specification represents the work of many organizations and individuals. The Kernel Model Language concept, as developed for use with SysML v2, is based on earlier work of the KerML Working Group, which was led by:

- Conrad Bock, US National Institute of Standards and Technology (NIST)
- Charles Galey, Lockheed Martin Corporation

- Bjorn Cole, Lockheed Martin Corporation

The primary authors of this specification document and the syntactic and semantic models described in it are:

- Ed Seidewitz, Model Driven Solutions
- Conrad Bock, US National Institute of Standards and Technology (NIST)
- Bjorn Cole, Lockheed Martin Corporation

The specification was formally submitted for standardization by the following organizations:

- 88Solutions Corporation
- Dassault Systèmes
- GfSE e.V.
- IBM
- INCOSE
- InterCax LLC
- Lockheed Martin Corporation
- MITRE
- Model Driven Solutions, Inc.
- PTC
- Simula Research Laboratory AS
- Thematix

However, work on the specification was also supported by over 120 people in over 60 other organizations that participated in the SysML v2 Submission Team (SST). The following individuals had leadership roles in the SST:

- Manas Bajaj, InterCax LLC (API and services development lead)
- Yves Bernard, Airbus (profile development co-lead)
- Bjorn Cole, Lockheed Martin Corporation (metamodel development co-lead)
- Sanford Friedenthal, SAF Consulting (SST co-lead, requirements V&V lead)
- Charles Galey, Lockheed Martin Corporation (metamodel development co-lead)
- Karen Ryan, Siemens (metamodel development co-lead)
- Ed Seidewitz, Model Driven Solutions (SST co-lead, pilot implementation lead)
- Tim Weilkiens, oose (profile development co-lead)

The specification was prepared using CATIA No Magic modeling tools and the OpenMBEE system for model publication (http://www.openmbee.org), with the invaluable support of the following individuals:

- Tyler Anderson, No Magic/Dassault Systèmes
- Christopher Delp, Jet Propulsion Laboratory
- Ivan Gomes, Jet Propulsion Laboratory
- Robert Karban, Jet Propulsion Laboratory
- Christopher Klotz, No Magic/Dassault Systèmes
- John Watson, Lightstreet consulting

# 7 Metamodel

## 7.1 Metamodel Overview

### 7.1.1 General

This clause specifies the syntax and part of the semantics of KerML (the complete semantics depends on model libraries, see below). It includes the following:

1. *Concrete syntax* specifies the how the language appears to modelers. They construct and review models shown according to the concrete syntax. The textual concrete syntax is based on a *lexical structure*, as defined in 7.1.2. Subclause 7.1.3 then describes the conventions for defining the grammar for the concrete syntax based on this lexical structure.
2. *Abstract syntax* (metamodel) specifies linguistic terms and relations between them (as opposed to library terms) that are expressed in concrete syntax. These omit purely visual aspects of concrete syntax, such as placement of shapes in graphical notation, or delimiters in textual notation, which do not affect what modelers are trying to express. Abstract syntax facilitates construction of tools that focus on how modelers use lingusitic terms, apart from how they appear visually. Concrete syntax is translated to abstract syntax by removing visual information (assuming both follow the specified syntaxes). Subclause 7.1.4 describes the conventions for defining abstract syntax.
3. *Semantics* specifies how to tell when actual or virtual systems conform to models in the way those systems are operated are constructed and during their operation (applies only to syntactically conformant models). As discussed in 6.1, a *core* subset of KerML abstract syntax is given a mathematical semantics. Semantics for the rest of KerML are specified by constraints on the use of KerML abstract syntax that require models to reuse of elements from the KerML model library, see 7.1.5.

The KerML metamodel is a taxonomy (repeated *layers* of specialization) of kinds of model elements (metaclasses), each of which includes the above facets. The taxonomy is divided into three layers (see 7.1.4), from general to specific:

1. *Root* includes the most general syntactic constructs for structuring models, such as elements, relationships, and packaging, see 7.2. These constructs have no semantics (in the sense of 6.1); this is added in specializations below.
2. *Core* includes the most general constructs that have semantics, based on *classification*, see 7.3. Some Core semantics is specified mathematically.
3. *Kernel* provides commonly needed modeling capabilities, such associations and behavior, see 7.4. Its additional semantics is specified entirely through model libraries.

### 7.1.2 Lexical Structure

#### 7.1.2.1 Lexical Structure Overview

The *lexical structure* of the KerML textual notation defines how the string of characters in an input text is divided into a set of *input elements*. Such input elements can be categorized as *whitespace*, *notes,* or *tokens.*

Lexical analysis is the process of converting an input text into a corresponding stream of input elements. After lexical analysis, whitespace and notes are discarded and only tokens are retained for the subsequent step of parsing. Lexical analysis for KerML is essentially the same as is done for the processing of any typical textual programming language.

#### 7.1.2.2 Line Terminators and White Space

```
LINE_TERMINATOR =
    "\n"
INPUT_CHARACTER =
    !("\n")
WHITE_SPACE =
    ' '| '\t' | '\f' | LINE_TERMINATOR
```

The input text can be divided up into lines separated by *line terminators*. A line terminator may be a single character (such as a line feed) or a sequence of characters (such as a carriage return/line feed combination). This specification does not require any specific encoding for a line terminator, but any encoding used must be consistent throughout any specific input text. Any characters in the input text that are not a part of line terminators are referred to as *input characters*.

A *white space* character is a space, tab, form feed or line terminator. Any contiguous sequence of white space characters can be used to separate tokens that would otherwise be considered to be part of a single token. It is otherwise ignored, with the single exception that a line terminator is used to mark the end of a single-line note (see 7.1.2.3).

#### 7.1.2.3 Notes and Comments

```
SINGLE_LINE_NOTE =
    '//' (!('\n'|'\r') !('\n'|'\r')*)? ('\r'? '\n')?;

MULTILINE_NOTE =
    '//*' -> '*/'

REGULAR_COMMENT =
    '/*' !'*' -> '*/';

DOCUMENTATION_COMMENT =
    '/**' -> '*/'
```

*Notes* and *comments* are used to annotate other elements of the input text. They have no computable semantics, but simply provide information useful to a human reader of the text. Notes and comments are lexically similar, but notes are not considered tokens and are, therefore, stripped from the input text and not parsed as part of the KerML concrete syntax. Comments, on the other hand, are parsed into Comment elements in the abstract syntax and are stored as part of the model represented by the input text. The lexical structure of comment text is described here. See 7.2.3 for the definition of the full syntax of Comment elements.

There are two kinds of notes:

1. A *single-line note* includes all the text from the initial characters "`//`" up to the next line terminator or the end of the input text (whichever comes first), except that "`//*`" begins a multi-line note rather than a single-line note.

   ```
   // This is a single-line note and will be ignored
   ```

2. A *multiline note* includes all the text from the initial characters "`//*`" to the final characters "`*/`".

```
//* This is a multiline note
           and will be ignored */
```

There are two kinds of comment text:

1. *Regular comment text* includes all the text from the initial characters "`/*`" to the final characters "`*/`", except that "`/**`" begins documentation comment text rather than regular comment text.

   ```
   /* This is the text for a regular Comment to be included in the model.

               It can be on a single line or multiple lines. */
   ```

2. *Documentation comment text* includes all the text from the initial characters "`/**`" to the final characters "`*/`". Regular comment text can be used to specify an sort of Comment, but documentation comment text is used solely to specify Documentation Comments (see 7.2.3).

   ```
   /** This is text for a Comment included as Documentation in the model. */
   ```

#### 7.1.2.4 Names

```
NAME =
    BASIC_NAME | UNRESTRICTED_NAME

BASIC_NAME =
    ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*

UNRESTRICTED_NAME =
    '\'' ( '\\' ('b'|'t'|'n'|'f'|'"'|"'"|'\\') | !('\\'|'\'') )* '\'' ;
```

Lexically, a name is a sequence of characters that is used to identify some model Element. This identification may be inherent to the element or relative to some *namespace* that provides a context for resolution of the name to the referenced Element. In either case, there are two kinds of names:

1. A *basic name* is one that can be lexically distinguish in itself from other kinds of tokens. The initial character of a basic name must be one of a lowercase letter, an uppercase letter or an underscore. The remaining characters of a basic name are allowed to be any character allowed as an initial character plus any digit. However, a reserved keyword may not be used as a name, even though it has the form of a basic name (see 7.1.2.7), including the Boolean literals **`true`** and **`false`**.

   ```
   Vehicle
   power_line
   ```

2. An *unrestricted name* provides a way to represent a name that contains any character. It is represented as a non-empty sequence of characters surrounded by single quotes. The characters within the single quotes may not include non-printable characters (including backspace, tab and newline). However, these characters may be included as part of the name itself through use of an escape sequence. In addition, the single quote character or the backslash character may only be included by using an escape sequence.

   ```
   '+'
   'circuits in line'
   'On/Off Switch'
   ```

An *escape sequence* is a sequence of two text characters starting with the backslash as an escape character, which actually denotes only a single character (except for the newline escape sequence, which represents however many characters is necessary to represent an end of line in a specific implementation—see 7.1.2.2). Table 1 shows the meaning of the allowed escape sequences.

**Table 1. Escape Sequences**

| Escape Sequence | Meaning |
|---|---|
| `\'` | Single Quote |
| `\"` | Double Quote |
| `\b` | Backspace |
| `\f` | Form Feed |
| `\t` | Tab |
| `\n` | Line Terminator |
| `\\` | Backslash |

### 7.1.2.5 Numeric Literals

```
DECIMAL_VALUE =
    '0'..'9' ('0'..'9')*

EXPONENTIAL_VALUE =
    DECIMAL_VALUE ('e' | 'E') ('+' | '-')? DECIMAL_VALUE
```

A *decimal value* represents a an exact decimal (base 10) representation of a natural number—that is, a non-negative integer. It consists of a sequence of one or more decimal digits (that is, characters "0" through "9"). A decimal value may specify a natural literal, or it may be part of the specification of a real literal (see 7.4.7.2.4). Note that a decimal literal does not include a sign, because negating a literal is an operator in the KerML Expression syntax.

```
0
1234
```

An *exponential value* is a decimal value followed by a base 10 exponential part delimited by the letter "e" or "E". An exponential value may be used in the specification of a real literal (see 7.4.7.2.4). Note that a decimal point and fractional part are not included in the lexical structure of an exponential value. They are handled as part of the syntax of real  literals.

```
5E3
2E-10
1E+3
```

**Submission Note.** For the revised submission, we may consider allowing other than decimal numeric literals, particularly the traditional binary, octal and hexadecimal.

### 7.1.2.6 String Values

```
STRING_VALUE =
    '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\\') | !('\\'|'"') )* '"'
```

A *string value* lexically delimits a sequence of characters to be included in a String literal value (see 7.4.7). The characters in the string value are surrounded by double quotes, within which escape characters resolve to their meaning as given in Table 2. The empty string is represented by a pair of double quote characters with no other characters intervening between them.

### 7.1.2.7 Reserved Words

A *reserved keyword* is a token that has the lexical structure do a basic name but is not actually be used as a basic name. The following keywords are so reserved in KerML.

**`about abstract all alias any as assoc behavior by binding bool class classifier comment composite conjugates conjugation connector datatype derived doc end element expr false feature first flow from function generalization import id in inout interaction inv language nonunique null of ordered out package port portion predicate private protected public readonly redefines relationship rep specializes step stream subclass subset subsets subtype succession then to true type typed`**

### 7.1.2.8 Symbols

*Symbols* are non-name tokens composed entirely of characters that are not alphanumeric. There are two kinds of symbols:

1. *Punctuation* symbols have no meaning themselves, but are used to allow unambiguous separation between other tokens that do have meaning.
2. *Operator* symbols are distinguished notations in the KerML Expression sublanguage (see 7.4.7) that map to particular library Functions.

Some symbols are made of of multiple characters that may themselves individually be valid symbol tokens. Nevertheless, a multi-symbol token is not considered a combination of the individual symbol tokens. For example, "`::`" is considered a single token, not a combination of two "`:`" tokens. Input characters shall be grouped from left to right to form the longest possible sequence of characters to be grouped into a single token. So "`a:::b`" would analyzed into four tokens: "`a`", "`::`", "`:`" and "`b`" (which, as it turns out, is not a valid sequence of tokens in the KerML textual concrete syntax).

Certain symbols can be used interchangeably with equivalent keywords. For convenience, these are referenced in the concrete syntax grammar using special lexical terminal names that match either the symbol or the corresponding keyword, as shown in Table 2.

**Table 2. Symbol/Keyword Pairs**

| Lexical Name | Symbol | Keyword |
|---|---|---|
| TYPED_BY | `:` | **typed by** |
| SPECIALIZES | `:>` | **specializes** |
| SUBSETS | `:>` | **subsets** |

| Lexical Name | Symbol | Keyword |
|---|---|---|
| REDEFINES | `:>>` | **redefines** |
| CONJUGATES | `~` | **conjugates** |

### 7.1.3 Concrete Syntax

The *grammar* definition for the KerML textual concrete syntax defines how lexical tokens for an input text (see 7.1.2) are grouped in order to construct an abstract syntax representation of a model (see 7.1.4). The concrete syntax grammar definition uses an Extended Back Naur Form (EBNF) notation (see Table 3) that includes further notations to describe how the concrete syntax maps to the abstract syntax (see Table 4).

Productions in the grammar formally result in the synthesis of classes in the abstract syntax and the population of their properties (see Table 5). Productions may also be parameterized, with the parameters typed by abstract syntax classes. Information passed in parameters during parsing allows a production to update the properties of the provided abstract syntax elements as a side-effect of the parsing it specifies. Some productions only update the properties of parameters, without synthesizing any new abstract syntax element.

**Table 3. EBNF Notation Conventions**

| | |
|---|---|
| **Lexical element** | `LEXICAL` |
| **Terminal element** | `'terminal'` |
| **Non-terminal element** | `NonterminalElement` |
| **Sequential elements** | `Element1 Element2` |
| **Alternative elements** | `Element1 \| Element2` |
| **Optional elements (zero or one)** | `Element ?` |
| **Repeated elements (zero or more)** | `Element *` |
| **Repeated elements (one or more)** | `Element +` |
| **Grouping** | `( Elements ... )` |

**Table 4. Abstract Syntax Synthesis Notation**

| | | |
|---|---|---|
| **Variable assignment** | `v = Element` | Assign the result of parsing the concrete syntax `Element` to the local variable `v`. |
| **Property assignment** | `x.p = Element` | Assign the result of parsing the concrete syntax `Element` to property `p` of the abstract syntax element denoted by `x`. |
| **List property construction** | `x.p += Element` | Add the result of parsing the concrete syntax `Element` to the list property `p` of the abstract syntax element denoted by `x`. |
| **Boolean property assignment** | `x.p ?= Element` | If the concrete syntax `Element` is parsed, then set the Boolean property `p` of the abstract syntax element denoted by `x` to true. |
| **Non-parsing assignment** | `{ v = value }`<br>`{ x.p = value }`<br>`{ x.p += value }` | Assign (or add) the given `value` to the variable `v` or property `x.p`, without parsing any input. |

**Table 5. Grammar Production Definitions**

| | | |
|---|---|---|
| **Synthetic production definition** | `NonterminalElement : AbstractSyntaxElement = ...` | Define a production for the `NonterminalElement` that synthesizes the `AbstractSyntaxElement`. |
| **Parameterized synthetic production definition** | `NonterminalElement (p1 : Type1, p2 : Type2, ...) : AbstractSyntaxElement = ...` | Define a production for the `NonterminalElement` that synthesizes the `AbstractSyntaxElement`, with the given parameters named `p1`, `p2`, .... The types of the parameters must be abstract-syntax classes. |
| **Parameterized updating production definition** | `NonterminalElement (p1 : Type1, p2 : Type2, ...) = ...` | Define a production for the `NonterminalElement` that does not synthesize any new abstract-syntax element, but updates properties of its parameters. (Such a production must have at least one parameter.) |

## 7.1.4 Abstract Syntax

The KerML metamodel is divided into three layers (see 7.1.1), each in a top-level package, as shown in Fig. 2. Each package publicly imports the one it depends on for more general metaelements, the Kernel package containing (as owned or imported members) all abstract syntax elements. Each package contains nested packages for the modeling areas it addresses.

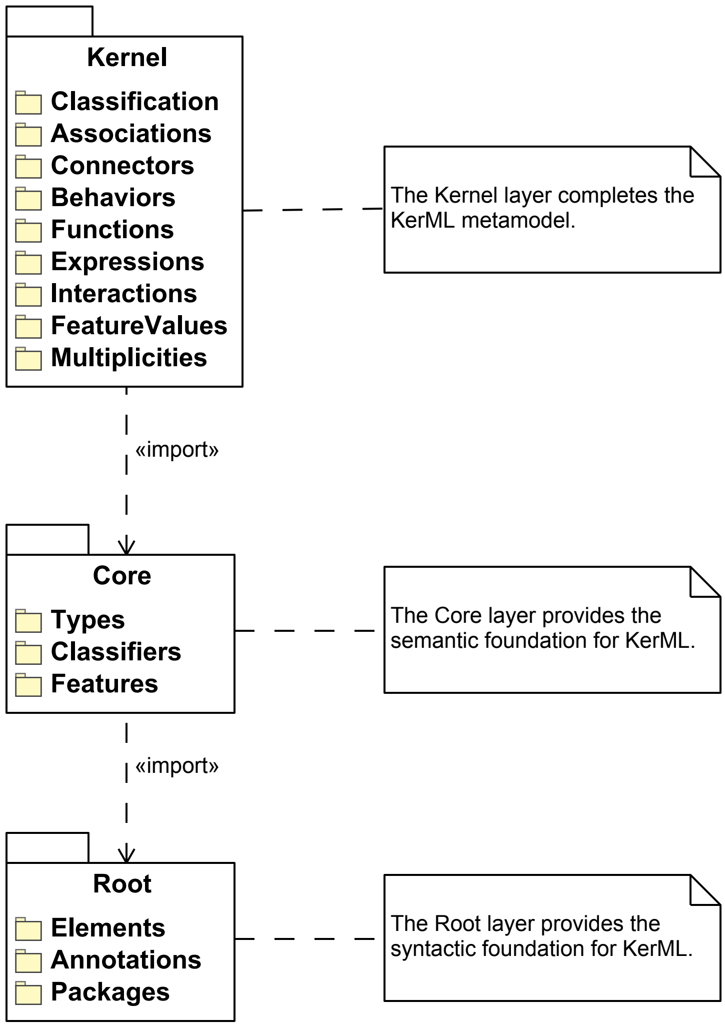


**Figure 2. KerML Syntax Layers**

Every metaclass in the KerML abstract syntax model is a direct or indirect subclass of the Element metaclass. Relationship is a particularly important direct subclass of Element. The KerML abstract syntax is designed so that models are represented as graphs of Elements connected by Relationships. The abstract syntax model rigorously follows the following convention: the only meta-associations that are not derived are those with Relationship metaclasses (that is Relationship or a subclass of it). All other associations between Elements are derived from such reified relationship classes.

Fig. 3 shows the complete generalization hierarchy of metaclasses in the KerML abstract syntax, excluding Relationship metaclasses other than Association and Connector (and their subclasses). Association and Connector (and their subclasses) are the only kind of Types that are also Relationships. Fig. 4 shows the hierarchy of Relationship metaclasses.

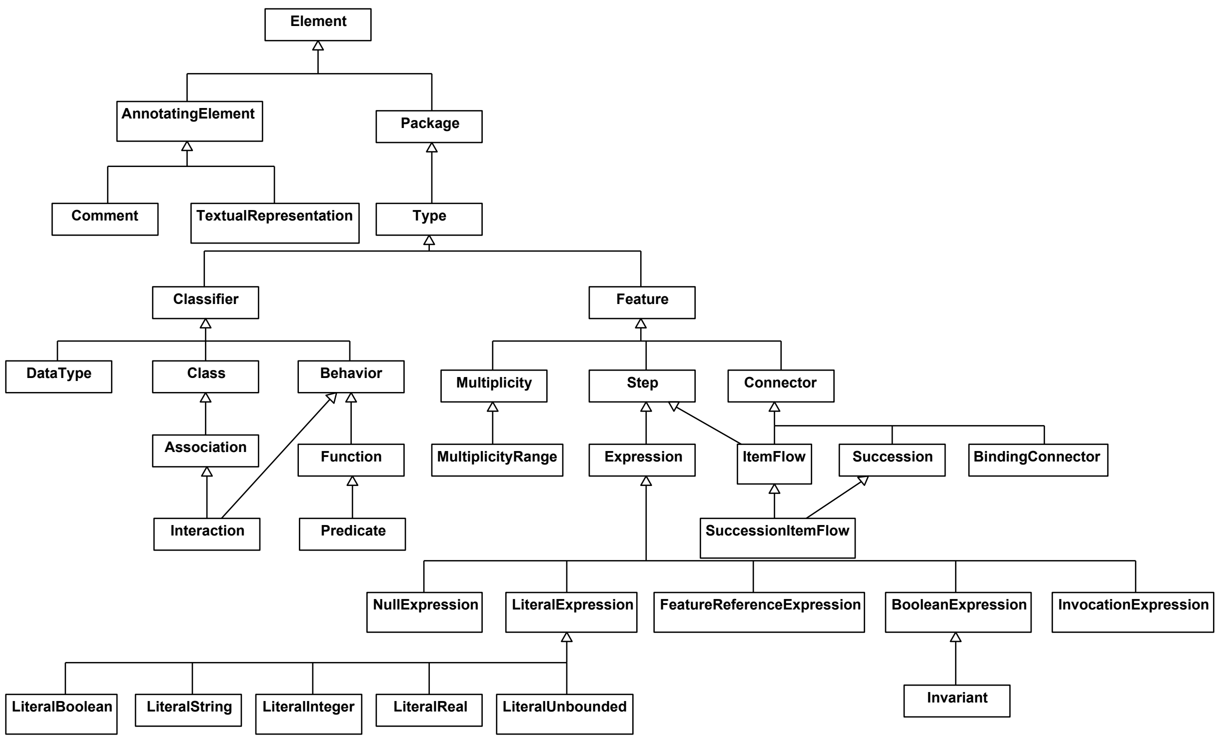


**Figure 3. KerML Element Hierarchy**

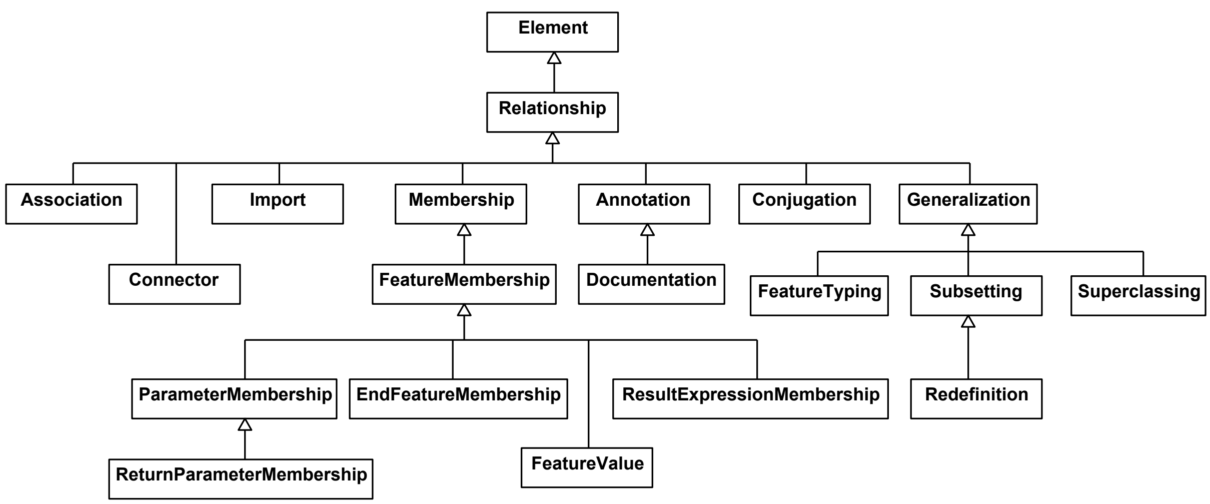


**Figure 4. KerML Relationship Hierarchy**

## 7.1.5 Semantics

KerML semantics is specified by a combination of mathematics and model libraries, as illustrated in 7.1.5. The left side of this diagram shows the abstract syntax packages corresponding to the three layers of the KerML metamodel. The right side shows the corresponding semantic layering.

The Root layer is purely syntactic and has no modeling semantics. The Core is grounded in mathematical semantics (based on 7.3.1.2), supported by the *Base* package from the Kernel Model Library (see 8.2). The Kernel layer is given semantics fully through its relationship to the Model Library (see Clause 8). The semantic specification for each Kernel sub-package summarizes constraints on Kernel abstract syntax elements that specify how the model library is used when models are constructed following the abstract syntax.

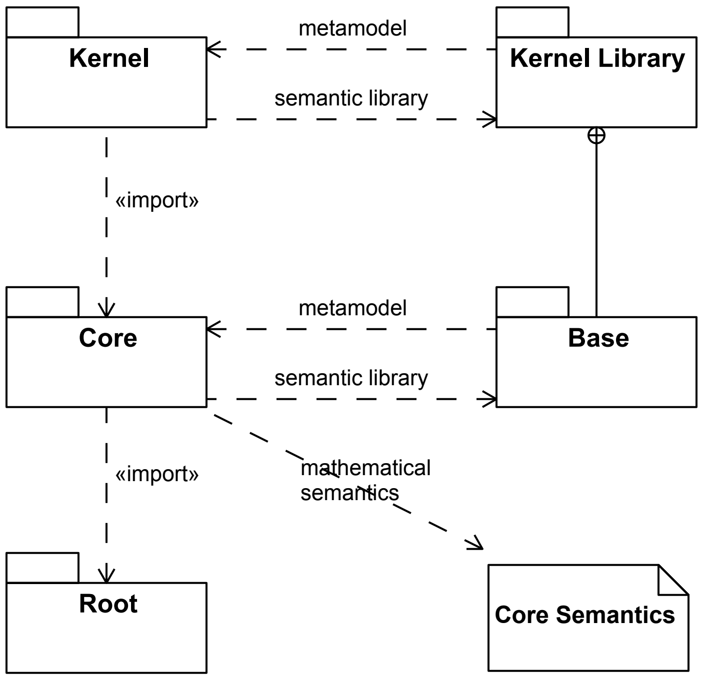


**Figure 5. KerML Semantic Layers**

# 7.2 Root

## 7.2.1 Root Overview

The Root layer contains the syntactic foundations of KerML. It includes constraints on the structure of models, but none of these affect the modeled systems as they are built or operate, that is, the elements have no semantics. This is added in the Core layer (see 7.3), which extends Root.

Root provides the most general syntactic capabilities of the language: Elements and Relationships between them, Annotations of Elements, and Membership of Elements in Packages. Packages also act as namespaces that can assign unique names to Package members, but support multiple aliases per Element. They also support Import of Elements from other Packages, enabling an Element to have a different name when imported.

## 7.2.2 Elements

### 7.2.2.1 Elements Overview

#### Identification

Elements are the constituents of a model. Every Element has an `identifier` that shall be a Universally Unique Identifier (UUID) (as specified in [UUID]). Generally, the properties of an Element can change over its lifetime, but the `identifier` shall not change after the Element is created.

The Element metaclass is the most general metaclass in the KerML abstract syntax. Element is *not* abstract, and a model may include instances of Element that are not instances of any other subclass of Element. Such an instance may be refined in later of the versions of the model into a more specific modeling construct, by dynamically changing its metaclass to a more specific specialization of Element (see [SMOF]). In general, the metaclass of an

Element may change over its lifetime, but all Element instances with the same `identifier` value shall be considered versions of the same constinuent model Element, regardless of their metaclass at any point in time.

An Element may also have additional identifiers, its `aliasIds`, which may be assigned for tool-specific purposes. This specification places no restrictions on the structure or uniqueness of `aliasIds` assigned by tools. It is a tool responsibility to manage any necessary uniqueness of such identifiers within or across models.

However, one of the `aliasIds`, the `humanId`, may be entered by the modeler. If given, the `humanId` for an Element has the lexical form of a name. However, an Element may be given different names relative to the namespaces provided by different Packages (see 7.2.4), while the `humanId` for an Element is the same in all contexts. Any `humanIds` of the `ownedElements` of a Package must be unique (see 7.2.4), but it is otherwise the responsibility of the modeler to maintain other structural or uniqueness properties for `humanIds` as appropriate to the model being created.

### Relationships

Some Elements represent Relationships between other Elements, known as the `relatedElements` of the Relationship. In general terms, a model is constructed as a graph structure in which Relationships form the edges connecting non-Relationship Elements constituting the nodes. However, since Relationships are themselves Elements, it is also possible in KerML for a Relationship to be a `relatedElement` in a Relationship and for there to be Relationships between Relationships.

The `relatedElements` of a Relationship are divided into `source` and `target` Elements. A Relationship is said to be *directed* from its `source` Elements to its `target` Elements. It is allowed for a Relationship to have only `source` or only `target` Elements. However, by convention, an *undirected* Relationship is usually represented as having only `target` Elements.

A Relationship shall have at least two `relatedElements`. A Relationship with exactly two `relatedElements` is know as a *binary* Relationship. A *directed binary* Relationship is a binary Relationship in which one `relatedElement` is the `source` and one is the `target`. Most specializations of Relationship in the KerML abstract syntax restrict the specialized Relationship to be a directed binary Relationship (the principal exceptions being Association and Connector and their further specializations).

### Ownership

One of the `relatedElements` of a Relationship may be the `owningRelatedElement` of the Relationship. If the `owningRelatedElement` of a Relationship is deleted from a model, then the Relationship shall also be deleted. Some of the `relatedElements` of a Relationship (which shall be distinct from the `owningRelatedElement`, if any) may also be designated as `ownedRelatedElements`. If a Relationship has `ownedRelatedElements`, then, if the Relationship is deleted from a model, all its `ownedRelatedElements` shall also be deleted.

The `ownedRelationships` of an Element are all those Relationships for which the Element is the `owningRelatedElement`. The `ownedElements` of an `Element` shall be all those `Elements` that are `ownedRelatedElements` of the `ownedRelationships` of the Element. The `owningRelationship` of an Element (if any) is the Relationship for which the Element is an `ownedRelatedElement`. An Element shall have no more than one `owningRelationship`. The owner of an Element (if any) shall be the `owningRelatedElement` of the `owningRelationship` of the Element.

The above deletion rules imply that, if an Element is deleted from a model, then all its `ownedRelationships` and `ownedElements` are also deleted. This may result in a further cascade of deletions until all deletion rules are satisfied. An Element that has no owner acts as the *root Element* of an *ownership tree structure,* such that all

Elements and Relationships in the structure are deleted if the root Element is deleted. Deleting any Element other than the root Element results in the deletion of the entire subtree below that Element.

It is a general design principle of the KerML abstract syntax that non-Relationship Elements are related only by reified instances of Relationships. All other meta-associations between Elements are derived from these reified Relationships. For example, the `owningRelatedElement/ownedRelationship` meta-association between an Element and a Relationship is fundamental to establishing the structure of a model. However, the `owner/ownedElement` meta-association between two Elements is derived, based on the Relationship structure between them.

### 7.2.2.2 Concrete Syntax

#### 7.2.2.2.1 Elements

```
Identification (e : Element, m : Membership) =
    ( 'id' e.humanId = NAME )? ( m.memberName = NAME )?

Element (m : Membership) : Element =
    'element' Identification(this, m) ElementBody(this)

ElementBody (e : Element) =
    ';' | '{' OwnedElement(e)* '}'

OwnedElement (e : Element) =
      e.ownedRelationship += OwnedRelationship(e)
    | e.documentation += OwnedDocumentation
    | e.ownedAnnotation += OwnedTextualRepresentationAnnotation
```

An Element in its simplest form, not representing any more specialized modeling construct, is notated using the keyword **`element`**. The declaration of an Element may also specify a humanId for it, as a lexical name preceded by the keyword **`id`**. Note that the notation does not have any provision or specifying the `identifier` or other `aliasIds` of an Element, since these are expected to be managed by the underlying modeling tooling.

```
element id e145;
```

If the Element is an `ownedMember` of a Package, then a `name` may also be given for the Element (after its `humanId`, if any). This `name` is actually the `memberName` of the Membership by which the Element is owned by the Package (see 7.2.4).

```
element id '1.2.4' MyName;
```

Note that it is not required to specify either a `humanId` or a `name` for an Element. However, unless at least one of these is given, it is not possible to reference the Element from elsewhere in the textual concrete syntax.

In addition to the declaration notated as above, the representation for an Element may include a *body,* which is a list of owned Elements delimited by curly braces `{ ... }`. It is a general principle of the KerML textual concrete syntax that the representation of owned Elements are nested inside the body of the representation of the owning Element. In this way, when the notation for the owning Element is removed in its entirety from the representation of a model, the owned Elements are also removed.

It is  possible to specify the following owned Elements as part of the body of an Element:

- Owned (generic) Relationships (see 7.2.2.2.2), using the keyword **`relationship`**. The containing Element becomes the `owningRelatedElement` and sole `source` for the Relationship with one or more other Elements identified as `target` Elements.
- Owned Documentation Comments (see 7.2.3), using the keyword **`doc`**. The containing Element becomes the `owningRelatedElement` for the Documentation Relationship to the Comment.
- Owned TextualRepresentations (see 7.2.3), using the keyword **`rep`** or **`language`**. The containing Element becomes the `ownedRelatedElement` for the Annotation Relationship to the TextualRepresentation.

```
element id A {
    doc /* Element A is related to element B. */
    relationship to B;
}
element id B {
    language "HTML"
        /* <a href="https://plm.elsewhere.com/part?id="1234"/> */
}
```

### 7.2.2.2.2 Relationships

```
Relationship (m : Membership) : Relationship =
    'relationship' Identification(this, m)
    RelationshipRelatedElements(this)
    RelationshipBody(this, m)

OwnedRelationship (e : Element) : Relationship =
    'relationship' Identification(this, m)
    'to' RelationshipTargetList(this)
    RelationshipBody(this)
    { source += e }

RelationshipRelatedElements (r : Relationship) =
      'from' RelationshipSourceList(r) ( 'to' RelationshipTargetList(r) )?
    | 'to' RelationshipTargetList(r)

RelationshipSourceList (r : Relationship) =
    RelationshipSource(r) ( ',' RelationshipSource(r) )*

RelationshipSource (r : Relationship) =
    r.source += [QualifiedName]

RelationshipTargetList (r : Relationship) =
    RelationshipTarget(r) ( ',' r.target += RelationshipTarget(r) )*

RelationshipTarget (r : Relationship) =
    r.target += [QualifiedName]

RelationshipBody (r : Relationship, m : Membership) =
    ';' | '{' RelationshipOwnedElement(r, m)* '}'

RelationshipOwnedElement (r : Relationship) =
      r.ownedRelatedElement += OwnedRelatedElement
    | r.documentation += OwnedDocumentation
    | r.ownedAnnotation += OwnedTextualRepresentationAnnotation

OwnedRelatedElement : Element =
      'element' ( humanId = NAME )? ElementBody
    | 'relationship' ( humanId = NAME )? RelationshipBody
```

A Relationship can be declared using the keyword **`relationship`**. As for a generic Element (see Elements above), a `humanId` and/or a `name` (if it is an ownedMember) may be specified for the Relationship. The (unowned) `source` Elements of the Relationship are then listed after the keyword **`from`**, while the target Elements are listed after the keyword **`to`**. It is allowable for a Relationship to have only `source` Elements or only `target` Elements, but there must be at least two Elements specified across the `source` and `target` lists (though some of the target Elements may be `ownedRelatedElements`, see below).

```
element id '1' A;
element id '2' B;
```

```
element id '3' C;
relationship id '4' R from '1' to B, C;
```

The top-level Elements of a model are implicitly declared within a *root* Package that provides a namespace for their `humanIds` and `names` (see 7.2.4). This allows for the identification of top-level Elements in the declaration of a Relationship, even without an explicit Package structure. However, when the model is organized into a Package structure, then Elements may be identified using qualified names according that structure (see 7.2.4 for the rules on the resolution of qualified names).

```
package P1 {
    element S;
}
package P2 {
    element T;
}
relationship from P1::S to P2::T;
```

A Relationship may have a body that specifies the following kinds of owned Elements of the Relationship:

- Owned (generic) Elements (see Elements above), using the keyword **`element`**. Such Elements become `ownedRelatedElements` of the containing Relationship (which are always target Elements).
- Owned (generic) Relationships, using the keyword, using the keyword **`relationship`**. Such Relationships become `ownedRelatedElements` of the containing Relationship, as for generic Elements.
- Owned Documentation Comments (see 7.2.3), using the keyword **doc**, as for a generic Element (see Elements above).
- Owned TextualRepresentations (see 7.2.3), using the keyword **rep** or **language**, as for a generic Element (see Elements above).

**Note.** The KerML concrete syntax does not provide any notation for a generic Relationship body to declare `ownedRelatedElements` of more specific kinds than listed above. A Package structure should be used instead to create a containment structure for more specific kinds of Elements (see 7.2.4).

To specify that a Relationship has an `owningRelatedElement`, use the nested owned Relationship notation (see Elements above).

```
element A;
element B {
    relationship to A {
        element C; // Owned related element
        relationship to B; // Relationship as owned related Element
    }
}
relationship R from A to B {
    doc /* This relationship has no owned related Elements. */
}
```

#### 7.2.2.3 Abstract Syntax

### 7.2.2.3.1 Overview

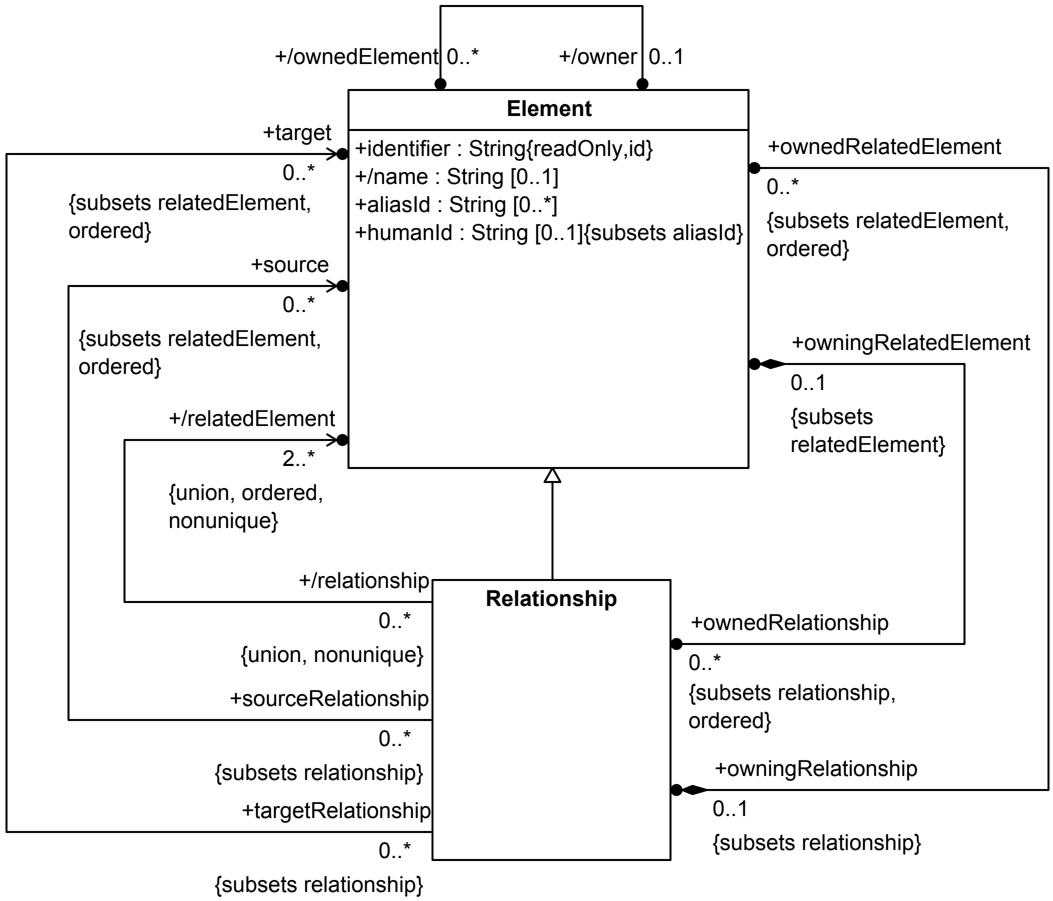


**Figure 6. Elements**

### 7.2.2.3.2 Element

**Description**

An Element is a constituent of a model that is uniquely identified relative to all other Elements. It can have Relationships with other Elements. Some of these Relationships might imply ownership of other Elements, which means that if an Element is deleted from a model, then so are all the Elements that it owns.

**General Classes**

No general classes.

**Attributes**

aliasId : String [0..*]

Various alternative identifiers for this Element. Generally, these will be set by tools, but one of them (the `humanId`), in particular, may be set by the modeler.

documentation : Documentation [0..*] {subsets ownedAnnotation}

The `ownedAnnotations` of this Element that are Documentation.

/documentationComment : Comment [0..*] {subsets ownedElement, annotatingElement}

The Comments that document this Element, derived as the `documentingComments` of the `documentation` of the Element.

humanId : String [0..1] {subsets aliasId}

An identifier for this Element that is set by the modeler. It is the responsibility of the modeler to maintain the uniqueness of this identifier within a model or relative to some other context. The `modeledId` essentially acts as an alias for an Element that is specifically tied to that Element, rather than being a name for it in the context of some explicit namespace.

identifier : String

The globally unique identifier for this Element. This is intended to be set by tooling, and it must not change during the lifetime of the Element.

/name : String [0..1]

The primary name of this Element. If the Element is owned by a Package, then its `name` is derived as the `memberName` of the `owningMembership` of the Element.

ownedAnnotation : Annotation [0..*] {subsets ownedRelationship, annotation}

The `ownedRelationships` of this Element that are Annotations.

/ownedElement : Element [0..*]

The Elements owned by this Element, derived as the `ownedRelatedElements` of the `ownedRelationships` of this Element.

ownedRelationship : Relationship [0..*] {subsets relationship, ordered}

The Relationships for which this Element is the `owningRelatedElement`.

/ownedTextualRepresentation : TextualRepresentation [0..*] {subsets ownedElement, textualRepresentation}

The `textualRepresentations` that are `ownedElements` of this Element.

/owner : Element [0..1]

The owner of this Element, derived as the `owningRelatedElement` of the `owningRelationship` of this Element, if any.

owningMembership : Membership [0..1] {subsets owningRelationship}

The `owningRelationship` of this Element, if that Relationship is a Membership.

/owningNamespace : Package [0..1] {subsets namespace}

The Package that is the owning namespace for this Element, derived as the `membershipOwningPackage` of the `owningMembership` of this Element, if any.

owningRelationship : Relationship [0..1] {subsets relationship}

The Relationship for which this Element is an `ownedRelatedElement`, if any.

**Operations**

No operations.

**Constraints**

elementName

[no documentation]

```
name = if owningNamespace = null then null
else owningNamespace.nameOf(self) endif
```

elementOwnedElements

[no documentation]

```
ownedElement = ownedRelationship.ownedRelatedElement
```

elementOwner

[no documentation]

```
owner = owningRelationship.owningRelatedElement
```

elementDocumentingComment

[no documentation]

```
documentingComment = documentation.documentingComment
```

### 7.2.2.3.3 Relationship

**Description**

A Relationship is an Element that relates two or more other Elements. Some of its `relatedElements` may be owned, in which case those `ownedRelatedElements` will be deleted from a model if their `owningRelationship` is. A Relationship may also be owned by another Element, in which case the `ownedRelatedElements` of the Relationship are also considered to be transitively owned by the `owningRelatedElement` of the Relationship.

The `relatedElements` of a Relationship are divided into `source` and `target` Elements. The Relationship is considered to be directed from the `source` to the `target` Elements. An undirected Relationship may have either all `source` or all `target` Elements.

A "relationship Element" in the kernel abstract syntax is generically any Element that is an instance of either Relationship or a direct or indirect specialization of Relationship. Any other kind of Element is a "non-relationship Element". It is a convention of the kernel abstract syntax that non-relationship Elements are *only* related via reified relationship Elements. Any meta-associations directly between non-relationship Elements must be derived from underlying reified Relationships.

**General Classes**

Element

**Attributes**

ownedRelatedElement : Element [0..*] {subsets relatedElement, ordered}

The `relatedElements` of this Relationship that are owned by the Relationship.

owningRelatedElement : Element [0..1] {subsets relatedElement}

The `relatedElement` of this Relationship that owns the Relationship, if any.

/relatedElement : Element [2..*] {ordered, nonunique, union}

The Elements that are related by this Relationship, derived as the union of the `source` and `target` Elements of the Relationship. Every Relationship must have at least two `relatedElements`.

source : Element [0..*] {subsets relatedElement, ordered}

The `relatedElements from which this Relationship is considered to be directed.`

target : Element [0..*] {subsets relatedElement, ordered}

The `relatedElements` to which this Relationship is considered to be directed.

**Operations**

No operations.

**Constraints**

No constraints.

## 7.2.3 Annotations

### 7.2.3.1 Annotations Overview

#### Annotations

An Annotation is a Relationship between an Element and an AnnotatingElement that provides additional information about the Element being annotated. Each Annotation is between a single AnnotatingElement and a single Element being annotated, but an AnnotatingElement may have multiple Annotation Relationships with different `annotatedElements`, and any Element may have multiple Annotations. The `annotatedElement` of an Annotation can optionally be the `owningRelatedElement` of the Annotation, in which case the `annotatedElement` is known as the `owningAnnotatedElement` and the Annotation is one of the `ownedAnnotations` of the `owningAnnotatedElement`.

Specific kinds of AnnotatingElements include Comments and TextualRepresentations.

**Submission Note.** It is expected that additional kinds of AnnotatingElements will be defined in the revised submission, such as for providing additional metadata for an Element. It is also expected that there will be a way for a modeler to add new kinds of AnnotatingElements as part of the language extension mechanism.

### Comments and Documentation

A Comment is an AnnotatingElement with a textual `body` that in some way describes its `annotatedElement`. A Comment that is related to its `annotatedElement` using the specialized Documentation Relationship has a special status of providing *documentation* for the `annotatedElement`. A Documentation Annotation shall be an `ownedAnnotation` of its annotatedElement. Further, the `documentingComment` of a Documentation shall be an `ownedRelatedElement` of the Documentation Relationship. This implies that the `documentationComments` of an Element (derived as the documentingComments of the owned documentation of the Element) are a subset of the `ownedElements` of the Element.

### Textual Representation

A TextualRepresentation is an AnnotatingElement whose textual `body` represents the `annotatedElement` in a given `language`. In particular, if the named `language` is machine-parsable, then the `body` text should be legal input text as defined for that `language`. The interpretation of the named language string shall be case insensitive. If the named `language` string matches one of the language names shown in Table 6 (without regard to case), then the body text shall be syntactically legal according to the specification shown in the table. Other specifications may define specific `language` strings, other than those shown in Table 6, to be used to indicate the use of languages from those specifications in KerML TextualRepresentations.

If the `language` of a TextualRepresentation is "`kerml`", then the `body` text shall be a legal representation of the `representedElement` in the KerML textual concrete syntax as defined in this specification. A conforming tool can use such a TextualRepresentation Annotation to record the original KerML concrete syntax text from which an Element was parsed. In this case, it is a tool responsibility to ensure that the `body` of the TextualRepresentation remains correct (or the Annotationis removed) if the annotated Element changes other than by re-parsing the `body` text.

For any other named `language`, the KerML specification does not define how the `body` text is to be semantically interpreted as part of the model being represented. In particular, a direct Element instance with a TextualAnnotation in a language other than KerML is essentially a semantically "opaque" Element specified in the other language. However, a conforming KerML tool may interpret such an element consistently with the specification of the named language.

**Table 6. Standard Language Names**

| Language Name | Specification |
|---|---|
| kerml | Kernel Modeling Language (this specification) |
| ocl | Object Constraint Language [OCL] |
| alf | Action Language for fUML [Alf] |

### 7.2.3.2 Concrete Syntax

#### 7.2.3.2.1 Comments

```
Comment (m : Membership, e : Element) : Comment =
    ( 'comment' Identification(this, m)
      'about' annotation += Annotation
      { ownedRelationship += annotation }
    ( ',' annotation += Annotation
      { ownedRelationship += annotation } )*
    | ( 'comment' Identification(this, m) )?
      annotation += ElementAnnotation(e)
      { ownedRelationship += annotation }
    )
    body = REGULAR_COMMENT

Annotation : Annotation ( =
    annotatedElement = [QualifiedName]

ElementAnnotation ( e : Element ) : Annotation =
    { annotatedElement = e }
```

The full declaration of a Comment begins with the keyword **`comment`**, optionally followed by a `humanId` and/or `name` (see 7.2.2). One or more `annotatedElements` are then identified for the Comment after the keyword **`about`**, indicating that the Comment has Annotation Relationships to each of the identified Elements. The `body` of the Comment is written lexically as regular comment text between "`/*`" and "`*/`" delimiters (see 7.2.2.1).

```
element A;
element B;
comment Comment1 about A, B
    /* This is the comment body text. */
```

If the Comment is an `ownedMember` of a Package (see 7.2.4), then the explicit identification of `annotatedElements` can be omitted, in which case the `annotatedElement` shall be implicitly the containing Package. Further, in this case, if no `humanId` or `name` is given for the Comment, then the **`comment`** keyword can also be omitted.

```
package P {
    comment C /* This is a comment about P. */

    /* This is also a comment about P. */
}
```

The actual `body` text of the Comment shall be extracted from the lexical regular comment token text as follows:

1. Remove the initial "`/*`" and final "`*/`" characters.
2. Remove any white space immediately after the initial "`/*`", up to and including the first line terminator (if any).
3. On each subsequent line of the text:
    1. Strip initial white space other than line terminators.
    2. Then, if the first remaining character is "`*`", remove it.
    3. Then, if the first remaining character is now a space, remove it.

For example, the lexical comment text in the following concrete syntax notation:

```
package CommentExample {
    /*
     * This is an example of multiline
     * comment text with typical formatting
     *     for readable display in a text editor.
     */
}
```

would result in the following body text in the Comment Element in the represented model:

```
This is an example of multiline
comment text with typical formatting
    for readable display in a text editor.
```

The body text of a Comment can include markup information (such as HTML), and a conforming tool may display such text as rendered according to the markup. However, marked up "rich text" for a Comment written using the KerML textual concrete syntax shall be stored in the Comment body in plain text including all mark up text, with all line terminators and white space included as entered, other than what is removed according to the rules above.

**Submission Note.** It is expected that the revised submission will provide a standard means for notating links to model elements from within Comments and a capability for the "transclusion" of certain textual information on model elements.

#### 7.2.3.2.2 Documentation

```
OwnedDocumentation : Documentation =
    documentingComment = DocumentationComment

DocumentationComment : Comment =
    'doc' ( 'id' humanId = Name )? body = REGULAR_COMMENT

PrefixDocumentation : Documentation =
    documentingComment = PrefixDocumentationComment

PrefixDocumentationComment : Comment =
    ( 'doc' ( 'id' humandId = Name )? )? body = DOCUMENTATION_COMMENT
```

A documentation Comment is notated similarly to a regular Comment (see 7.2.3.2.1), but using the keyword **doc** rather than **comment**. Since a documentation Comment is always an ownedElement of its annotatedElement, the notation of a documentation Comment is always nested within the notation of its owning Element, so there is no need to explicitly identify the annotatedElement. Further, since a documentation Comment is always owned via its Documentation Relationship, it cannot be an ownedMember of a Package and therefore cannot have a name specified for it. However, it can optionally be given a humanId (or separately given an alias via a non-owning Membership Relationship—see 7.2.4).

```
element X {
    doc id X_Comment
        /* This is a documentation comment about X. */
    doc /* This is more documentation about X. */
}
```

If the Element being documented is the member of a Package, then a special notation can be used in which lexical documentation comment text (see 7.2.2.1) is place immediately *before* the notation of the Element being documented. If no `humanId` is specified for a documentation Comment of this form, then the keyword **`doc`** can also be omitted.

```
package P {

    /** This is a documentation comment about Q. */
    package Q;

}
```

Documentation comment text shall be processed following the same rules as for regular comment text (see above), except that the initial "`/**`" characters are removed, rather than just "`/*`".

#### 7.2.3.2.3 Textual Representation

```
OwnedTextualRepresentationAnnotation : Annotation =
    ownedRelatedElement += OwnedTextualRepresentation(this)

OwnedTextualRepresentation (a : Annotation) : TextualRepresentation =
    ( 'rep' ( humanId = NAME )? )?
    'language' language = STRING_VALUE body = REGULAR_COMMENT
    { annotation += a }

TextualRepresentation (m : Membership, e : Element) : TextualRepresentation =
    ( 'rep' Identification(this, m)
      'about' annotation += Annotation
    | ( 'rep' Identification(this, m) )?
      ElementAnnotation(e)
    )
    'language' language = STRING_VALUE body = REGULAR_COMMENT
```

A TextualRepresentation is notated similarly to a regular Comment (see 7.2.3.2.1), but with the keyword **`rep`** used instead of **`comment`**. Similarly to a Comment, the **`about`** keyword can be used to specify the `representedElement` for the TextualRepresentation. However, only one `representedElement` may be identified for a TextualRepresentation. If the TextualRepresentation is an `ownedMember` of a Package (see 7.2.4), then, if the `representedElement` is not identified explicitly, it shall by default be the containing Package. A TextualRepresentation declaration must also specify the `language` as a literal string following the keyword **`language`**. If the TextualRepresentation has no `humanId`, `name` or explicit `representedElement`, then the **`rep`** keyword can also be omitted.

```
class C {
    feature x: Real;
    inv x_constraint;
    rep inOCL about x_constraint language "ocl"
        /* self.x > 0.0 */
}
behavior setX(c : C, newX : Real) {
    language "alf"
        /* c.x = newX;
```

```
            * WriteLine("Set new x");
            */
}
```

The lexical comment text given for a TextualRepresentation shall be processed as for regular comment text (see above), and it is the result after such processing that is the TextualRepresentation `body` expected to conform to the named `language`.

**Note.** Since the lexical form of a comment is used to specify the TextualRepresentation `body`, it is not possible to include comments of a similar form in the `body` text.

**Submission Note.** The revised submission may include a means to allow nested comments.

### 7.2.3.3 Abstract Syntax

#### 7.2.3.3.1 Overview

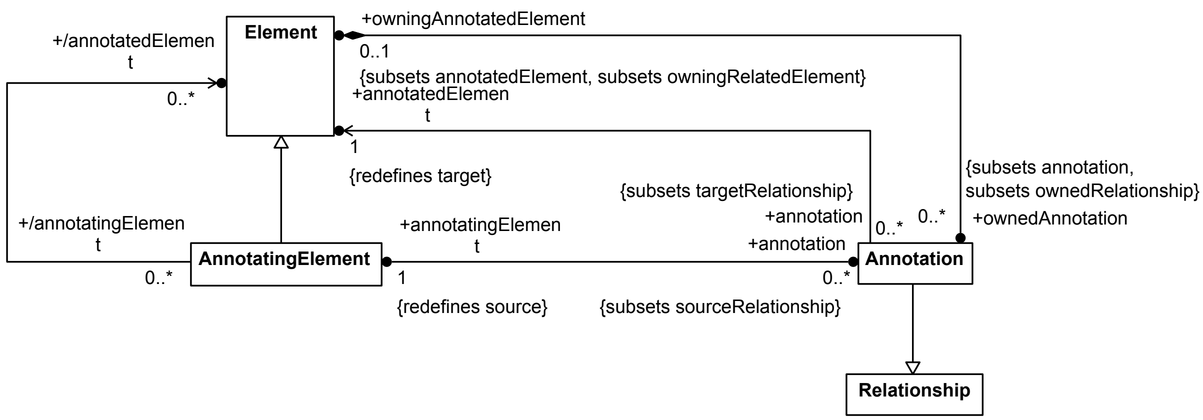


**Figure 7. Annotation**

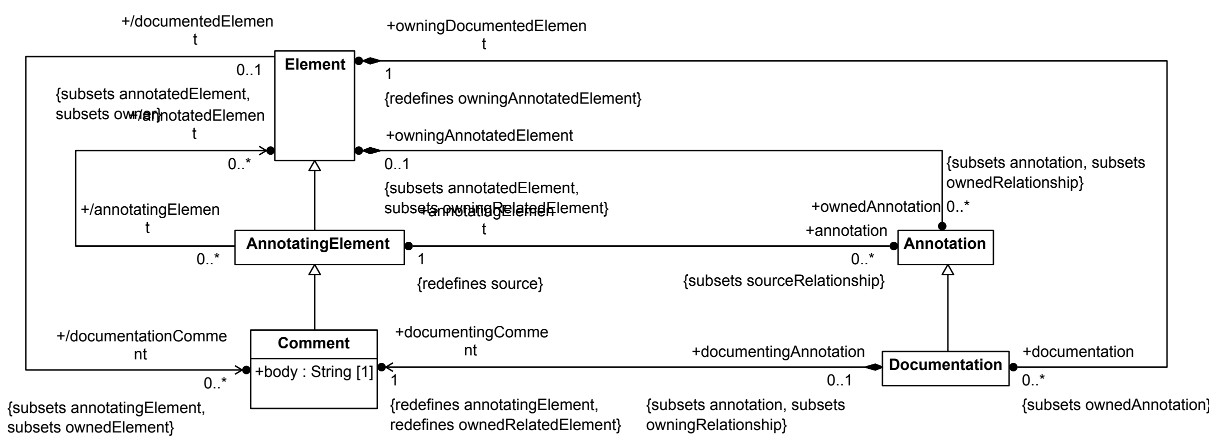


**Figure 8. Comments**

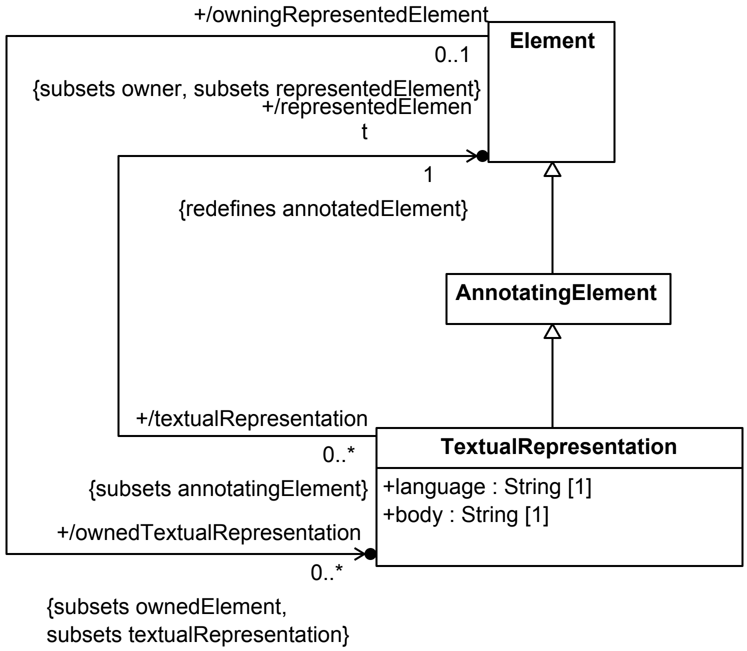


**Figure 9. Textual Representation**

### 7.2.3.3.2 AnnotatingElement

**Description**

An AnnotatingElement is an Element that provides additional description of or metadata on some other Element. An AnnotatingElement is attached to its `annotatedElement` by an Annotation Relationship.

**General Classes**

Element

**Attributes**

/annotatedElement : Element [0..*]

The Elements that are annotated by this AnnotatingElement, derived as the `annotatedElements` of the `annotations` of this AnnotatingElement.

annotation : Annotation [0..*] {subsets sourceRelationship}

The Annotations that relate this AnnotatingElement to its `annotatedElements`.

**Operations**

No operations.

**Constraints**

annotatingElementAnnotatedElement

[no documentation]

```
annotatedElement = annotation
.annotatedElement
```

#### 7.2.3.3.3 Annotation

**Description**

An Annotation is a Relationship between an AnnotatingElement and the Element that is annotated by that AnnotatingElement.

**General Classes**

Relationship

**Attributes**

annotatedElement : Element {redefines target}

The Element that is annotated by the `annotatingElement` of this Annotation.

annotatingElement : AnnotatingElement {redefines source}

The AnnotatingElement that annotates the `annotatedElement` of this Annotation.

owningAnnotatedElement : Element [0..1] {subsets annotatedElement, owningRelatedElement}

The `annotatedElement` of this Annotation, when it is also its `owningRelatedElement`.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.2.3.3.4 Comment

**Description**

A Comment is AnnotatingElement whose `body` in some way describes its `annotatedElements`.

**General Classes**

AnnotatingElement

**Attributes**

body : String

The annotation text for the Comment.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.2.3.3.5 Documentation

**Description**

Documentation is an Annotation whose `annotatingElement` is a Comment that provides documentation of the `annotatedElement`. Documentation is always an `ownedRelationship` of its `annotedElement`.

**General Classes**

Annotation

**Attributes**

documentingComment : Comment {redefines annotatingElement, ownedRelatedElement}

The Comment, which is owned by the Documentation Relationship, that documents the `owningDocumentedElement` of this Documentation.

owningDocumentedElement : Element {redefines owningAnnotatedElement}

The `annotatedElement` of this Documentation, which must own the Relationship.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.2.3.3.6 TextualRepresentation

**Description**

A TextualRepresentation is an AnnotatingElement that whose `body` represents the `representedElement` in a given `language`. The named `language` can be a natural language, in which case the `body` is an informal representation, or an artifical language, in which case the `body` is expected to be a formal, machine-parsable representation.

**General Classes**

AnnotatingElement

**Attributes**

body : String

A textual representation of the `representedElement` in the given `language`.

language : String

The natural or artifical language in which the `body` text is written.

/representedElement : Element {redefines annotatedElement}

The Element represented textually by this TextualRepresentation, which is its single `annotatedElement`.

**Operations**

No operations.

**Constraints**

No constraints.

## 7.2.4 Packages

### 7.2.4.1 Packages Overview

#### Memberships

A Package is an Element that contains other Elements via Membership Relationships with those Elements. The Package that is the `source` of a Membership Relationship shall also be its `owningRelatedElement`, known as the `membershipOwningPackage` of the Membership. The `target` of a Membership can be any kind of Element, known as the `memberElement` of the Membership.

The Memberships for which a Package is the `membershipOwningPackage` are the `ownedMemberships` of the Package. If the `memberElement` of an `ownedMembership` is an `ownedRelatedElement` of the Membership, then it is an `ownedMemberElement` of the Membership and an `ownedMember` of the Package. If an Element is the `ownedMemberElement` of a Membership, then that Membership is known as the `owningMembership` of the Element.

A Package may also have Import Relationships to other Packages. The Package that is the `source` of an Import Relationship shall also be its `owningRelatedElement`, known as the `importOwningPackage` of the Import. The Package that is the `target` of an Import Relationship is known as the `importedPackage` of the Import. The `importOwningPackage` of an Import shall be different than its `importedPackage`.

The visible Memberships of the `importedPackage` of an Import shall become `importedMemberships` of the `importOwningPackage`. The *visible* Memberships of a Package shall comprise at least the follow:

- All `ownedMemberships` of the Package with `visibility = public`.
- All `importedMemberships` of the Package that are derived from Import Relationships with `visibility = public`.

Subclasses of Package my define additional Memberships to be included in the set of visible Memberships of that kind of Package (for instance, the visible Memberships of a Type also include the public `inheritedMemberships` of the Type—see 7.3.2).

A Package can also have `ownedMemberships` for which the `memberElement` is *not* owned. The union of the set of unowned `memberElements` of `ownedMemberships` and the set of `memberElements` (owned or unowned) of `importedMemberships` my be referred to as the complete set of *imported members* of the Package. The `members` of a Package comprise at least its `ownedMembers` and the complete set of its imported members.

A *root Package* is a Package that has no owner. The `ownedElements` of a root Package are known as *top-level Elements*. Any Element that is not a root Package shall have an `owner` and, therefore, must be in the ownership tree of a top-level Element of some root Package.

**Note.** The set of all Elements owned directly or indirectly by a root Package may be considered to be the representation of a single "model", though this term is not formally defined within KerML.

### Namespaces

A Package also acts as a *namespace* for its `members`, such that each `member` can optionally be given one or more names *relative to* that Package. The names of a `member` of a Package shall consist of the `memberNames` specified for all the Memberships by which the `member` Element is related to the Package. Note that the same Element may be related to a Package by multiple Memberships, allowing the Element to have multiple, different names relative to that Package.

The `name` property of an Element is derived as the `memberName` of the `owningMembership` of the Element. All other names given to an Element are termed *aliases* for the Element.

The names of all the `ownedMembers` of a Package shall be distinct from each other. Further, if the `memberName` of any visible Membership of an `importedPackage` conflicts with the name of any of `ownedMember` of the `importOwningPackage`, or with the `memberName` of any visible Membership of the `importedPackage` of any other Import, then that Membership shall be considered *hidden,* and it shall *not* be included in the set of `importedMemberships` of the `importOwningPackage`.

As a result of the above rules, the `memberNames` of all `ownedMemberships` and `importedMemberships` will always be distinct from each other. Any subclass of Package that adds further kinds of Memberships (e.g., `inheritedMemberships` of Types—see 7.3.2) shall maintain the property that the `memberNames` of all `memberships` of a Package are distinct from each other.

**Submission Note.** The current rules for Membership distinguishibility in a Package require that all memberNames be distinct from each other. It is expected that this will be loosened in the revised submission to allow overloading of behavioral Elements with the same name when these can be distinguished by having different parameter signatures.

Package is an Element that may itself be a named `member` of another Package. A *qualified name* of a named Package `member` includes both its *unqualified* `memberName` and the name of its containing Package, which may or may not be itself qualified. A qualified name of an Element has the form of a list of the `memberNames` of Packages each relative to the previous one, followed by the unqualified `memberName` of the Element in the final Package. Since Packages may themselves have aliases, it is possible for there to be multiple qualified names for an Element even if it does not itself have aliases. On the other hand, if a Package does not have any name, then its `members` will have no qualified names, even if they are themselves named.

Since a root Package cannot be contained in any other Package, it cannot have a `name`, at least as given within the KerML language. The namespace of the root Package is known as the *root namespace,* which includes the names of all the members of the root Package with non-null memberNames. Any qualified name of an Element relative to a root Package always begins with the name of a member of the root Package, without regard to the (nameless) root Package itself.

**Note.** While a root Package cannot be given a name within KerML, it is expected that it would be named by what ever tooling or repository is used to manage KerML models. For the purposes of document-based model interchange, a root Package is the top-level element that can be interchanged as a single document (see Clause 9).

### 7.2.4.2 Concrete Syntax

#### 7.2.4.2.1 Packages

```
RootPackage : Package =
    PackageBodyElement(this)*

Package (m : Membership) : Package =
    PackageDeclaration(this, m) PackageBody(this)

PackageDeclaration (p : Package, m : Membership) : Package =
    'package' Identification(p, m)

PackageBody (p : Package) =
    ';' | '{' PackageBodyElement(p)* '}'
```

The *declaration* of a Package gives its identification, while the *body* of a Package specifies its contents.

The declaration of a root Package is implicit and no identification of it is provided in the KerML textual notation. Instead, the body of a root package (i.e., a KerML "model") is given simply by the list of representations of its top-level elements, typically in a single textual document.

```
doc /* This is a model notated in KerML concrete syntax. */
element A {
    relationship B to C;
}
class C;
datatype D;
feature f: C;
package P;
```

A Package that is not a root Package, and does not represent any more specialized modeling construct (such as a Type—see 7.3.2) is declared using the keyword **`package`**, optionally followed by a `humanId` and/or `name` (see 7.2.2). The body of the Package is notated as a list of representations of the content of the package delimited between curly braces `{...}`. If the Package is empty, then the body may omitted and the declaration ended instead with a semicolon.

```
package id '1.1' P1; // This is an empty package.
package id '1.2' P2 {
    doc /* This is an example of a package body. */
    class C;
    datatype D;
    feature f : C;
    package P3; // This is a nested package.
}
```

#### 7.2.4.2.2 Package Bodies

```
PackageBodyElement (p : Package) =
      p.documentation += OwnedDocumentation
    | p.ownedMembership += PackageMember(p)
    | p.ownedImport += PackageImport

PackageMember (p : Package) : Membership
    ( documentation += PrefixDocumentation )*
    ( visibility = BasicVisibilityIndicator )?
    ( NonFeaturePackageMember(this, p)
    | FeaturePackageMember(this) )

NonFeaturePackageMember (m : Membership, p : Package) =
      m.ownedMemberElement = NonFeatureElement(m, p)
    | ( 'alias' | 'import' ) m.memberElement = [QualifiedName]
      ( 'as' m.memberName = NAME )? ';'

FeaturePackageMember (m : Membership) =
    m.ownedMemberElement = FeatureElement(m)

PackageImport : Import =
    ( documentation += PrefixDocumentation )*
    ( visibility = BasicVisibilityIndicator )?
    'import' importedPackage = [QualifiedName] '::' '*' ';'

BasicVisibilityIndicator : VisibilityKind =
    'public' | 'private'
```

Declaring an Element within the body of a Package denotes that the Element is an `ownedMember` of the Package—that is, that there is an `ownedMembership` of the Package with the Element as its `ownedMemberElement`. The name given for the Element (if any) becomes the `memberName` of the Membership. The `visibility` of the Membership can also be specified by placing the keyword **`public`** or **`private`** before the Element declaration. If no visibility is specified, the default is **`public`**.

```
package P {
    public class C;
    private datatype D;
    feature f : C; // public by default
}
```

An alias for an Element is declared using the keyword **`alias`** followed by a qualified name (see below) identifying the Element, with the alias name given after the keyword **`as`**. This denotes an `ownedMembership` of the containing Package, with the identified Element as an unowned `memberElement`. The `visibility` of the Membership can be specified as for an `ownedMember`. The identified Element need not be an `ownedMember` of the Package containing the alias declaration, in which case the explicit alias name may be omitted, with the `name` of the Element then being the implicit alias name within the aliasing Package.

The keyword **`import`** can be used instead of **`alias`**. Typically, **`import`** is used when the aliased Element is not an `ownedMember` of the aliasing Package, while **`alias`** is used when giving another name to an Element within the same Package. However, the two forms are actually completely equivalent in meaning.

```
package P1 {
    class A;
    private alias A as B;
    public import P::C as CC;
    import P::D;
}
```

An `ownedImport` of a Package is denoted using the keyword **`import`** followed by a qualified name (see below) identifying the `importedPackage`, followed by "`::*`". The visibility of the Import can be specified by placing the keyword **`public`** or **`private`** before the Import declaration. If no visibility is specified, the default is **`public`**.

```
package P2 {
    import P::*;
    private import P1::*;
}
```

A regular Comment (see 7.2.3.2) declared within a Package body also becomes an `ownedMember` of the Package. If no `annotatedElements` are specified for the Comment, then, by default, the Comment is considered to be about the containing Package.

```
package P3 {
    class A;
    comment Comment1 about A
        /* This is a comment about class A. */

    comment Comment 2
        /* This is a comment about package P3. */

    /* This is also a comment about package P3. */
}
```

A Documentation Comment declared within a Package body (see 7.2.3.2), however, is *not* an `ownedMember` of the Package. Instead, if it is a regular Comment, then it is owned via a Documentation Relationship by the containing Package. If it is a prefix Comment, then it is owned via a Documentation Relationship with the next Membership or Import declared in the Package lexically after the Comment.

```
package P4 {
    doc P4_Doc
        /* This is documentation about package P4. */

    /** This is documentation about member B. */
    /** This is more documentation about member B. */
    private class B;

    /** This is documentation about alias B1. */
    public alias B as B1;

    /** This is documentation about the import of P3. */
    import P3::*;
}
```

#### 7.2.4.2.3 Packaged Elements

```
NonFeatureElement (m : Membership, p : Package) : Element =
       Element(m)
     | Relationship(m)
     | Comment(m, p)
     | TextualRepresentation(m, p)
     | Package(m)
     | Type(m)
     | Classifier(m)
     | Class(m)
     | DataType(m)
     | Association(m)
     | Interaction(m)
     | Behavior(m)
     | Function(m)
     | Predicate(m)
     | Generalization(m)
     | Conjugation(m)
     | Superclassing(m)
     | FeatureTyping(m)
     | Subsetting(m)
     | Redefinition(m)

FeatureElement (m : Membership) : Feature =
       Feature(m)
     | Step(m)
     | Expression(m)
     | BooleanExpression(m)
     | Invariant(m)
     | Connector(m)
     | BindingConnector(m)
     | Succession(m)
     | ItemFlow(m)
     | SuccessionItemFlow(m)
```

A Package body can contain any kind of Element that can be represented in the KerML notation. These are syntactically divided into two sets: Feature Elements and non-Feature Elements. Feature Elements include Feature, as defined in the Core (see 7.3.4, and the various specialized kinds of Features defined in the Kernel (see 7.4). Non-Feature Elements include all constructs defined in the Root (see 7.2), Type and Classifier as defined in the Core (see 7.3.2 and 7.3.3), and the various specialized kinds of Classifiers defined in the Kernel (see 7.4). This division is convenient because, in the Core, Feature Elements are related to Types using a specialized FeatureMembership Relationship, while non-Feature Elements are related to Types using the same generic Membership Relationship used with non-Type Packages.

#### 7.2.4.2.4 Name Resolution

```
QualifiedName =
    NAME ( '::' NAME )*
```

A qualified name is notated as a sequence of *segment names* separated by "`::`" punctuation. An *unqualified* name can be considered the degenerate case of a qualified name with a single segment name. A qualified name is used in the KerML textual concrete syntax to identify an Element that is being referred to in the representation of another Element. A qualified name used in this way does not appear in the corresponding abstract syntax—instead, the abstract syntax representation contains an actual reference to the identified Element. *Name resolution* is the process of determining the Element that is identified by a qualified name.

Qualified name resolution uses the namespaces provided by Package memberships in order to map simple names to named Elements. Every Package other than a root Package is nested in a containing Package called its `owningNamespace`. A root Package has an implicit containing namespace known as its *global namespace.* The global namespace for a root Package includes all the visible Memberships of all other root Packages that are *available* to the first Package, which shall include at least all the KerML Model Libraries (see Clause 8). A conforming tool can provide means for making additional Packages available to a root Package, but this specification does not define any standard mechanism for doing so.

An Element considered to be *directly contained* in a Package if it is an ownedElement of the Package or if it is indirectly owned by the Package without any other intervening Package (e.g., if the Element is an ownedRelatedElement of a Relationship that is not a Membership but is an ownedMember of the Package). The namespace of a Package defines a mapping from names to Elements directly contained in the Package, know as the *local resolution* of those names.

1. For each Element that is directly contained in a Package, the `humanId` of the Element locally resolves to that Element.
2. For each `membership` of a Package, the `memberName` of the Membership locally resolves to the `memberElement` of the Membership.

**Note.** If the Package is well formed, then there can be at most one Element that locally resolves to any given name.

The *visible resolution* of a name restrictes the `memberships` in the second step to those that are visible outside the Package. Note that resolution of `humanIds` is not restricted by visibility.

In general, the *full resolution* of a simple name relative to a Package namespace then proceeds as follows:

1. If the name locally resolves to an Element directly contained in the Package, then it fully resolves to that Element.
2. If there is no such Element, then:
    - If the Package is *not* a root Package, then the name resolution continues with the `owningNamespace` of the Package.
    - If the Package *is* a root Package, then the name resolution continues with the global namespace.

The resolution of a simple name in the global namespace proceeds as follows:

1. If there is a Membership in the global namespace that has an `ownedMemberElement` that has a `humanId` equal to the simple name, then the name resolves to that Element.
2. If there is a Membership in the global namespace that has a `memberName` equal to the simple name, then the name resolves to the `memberElement` of that Membership.
3. If there is no such Membership, then the name has no resolution.

**Note.** It is possible that there will be more than one Membership that resolves to a given simple name. In this case, one of these Memberships is chosen for the resolution of the name, with `humanId` resolution having priority over `memberName` resolution, but with which one is chosen not otherwise determined by this specification.

A qualified name is always used to identify an Element that is a `target` Element of some Relationship. The *context* Package is the nearest Package that directly or indirectly owns that Relationship. The *local namespace* for resolving the qualified name is then determined as follows:

- If the context Relationship is *not* a Membership or an Import, then the local namespace is the context Package.
- If the context Relationship *is* a Membership or an Import, then
    - If the context Package is *not* a root Package, then the local namespace is the `owningNamespace` of the context Package.
    - If the context Package *is* a root Package, then the local namespace is the global namespace for the context Package.

**Note.** Membership and Import Relationships are treated as a special case in order to avoid possible infinite recursion in the name resolution process.

The resolution of a qualified name begins with the full resolution of its first segment name with respect to the local namespace for the qualified name. If the qualified name has only one segment name, then the qualified name resolves to the resolution of its first segment name. Otherwise, each segment name of the qualified name, other than the last, must resolve to a Package that is the visible resolution of the name relative to the Package identified by the previous segment. The qualified name then resolves to the resolution of its last segment name.

#### 7.2.4.3 Abstract Syntax

### 7.2.4.3.1 Overview

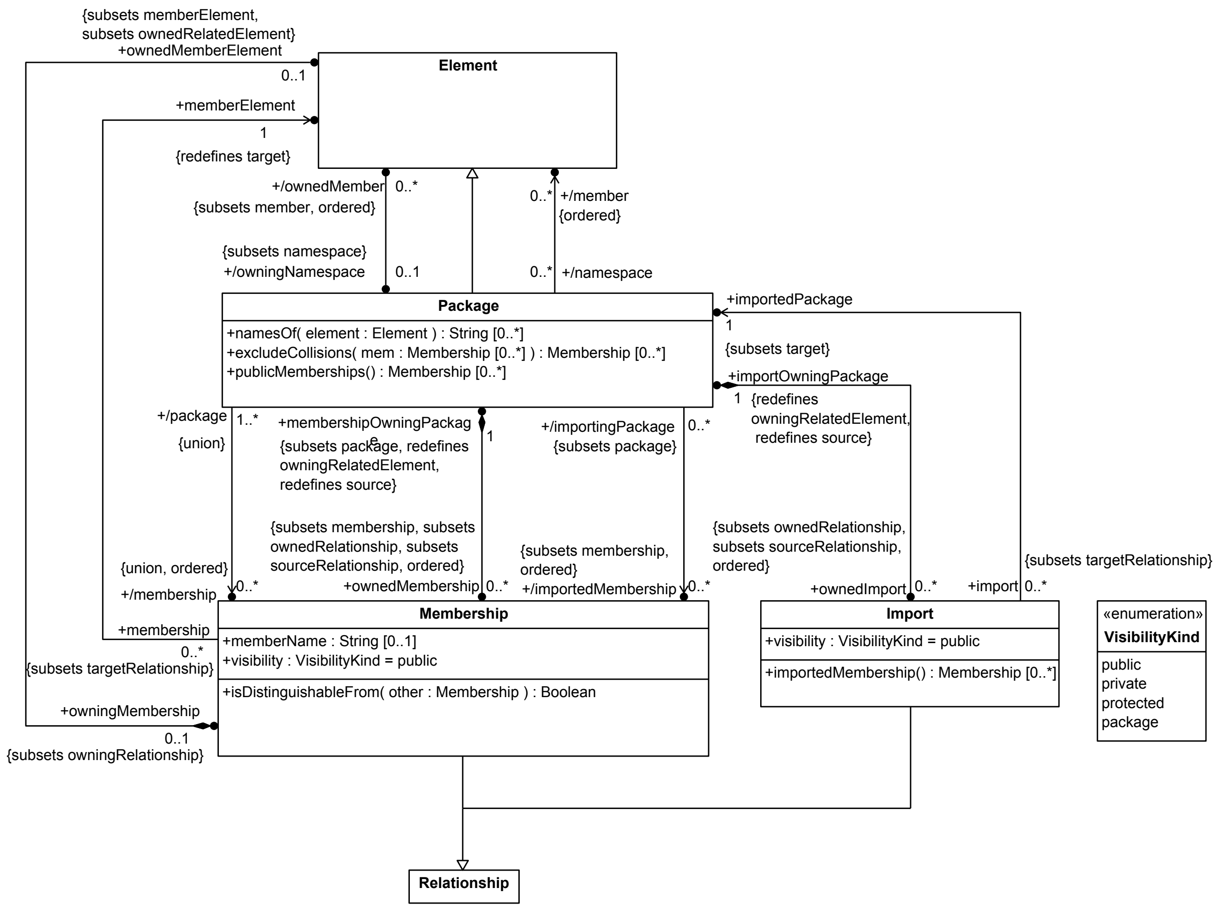


**Figure 10. Packages**

### 7.2.4.3.2 Import

**Description**

An Import is a Relationship between an `importOwningPackage` and an `importedPackage` in which the visible member Elements of the `importedPackage` become imported `members` of the `importOwningPackage`. An Import may be *public,* in which case the imported `members` are "re-exported" as publicly visible `members` of the `importOwningPackage`, or it may be *private,* in which case the imported `members` are private to the `importOwningPackage`.

**General Classes**

Relationship

**Attributes**

importedPackage : Package {subsets target}

The Package whose visible `members` are imported by this Import.

importOwningPackage : Package {redefines source, owningRelatedElement}

The Package into which `members` are imported by this Import, which must be the `owningRelatedElement` of the Import.

visibility : VisibilityKind

Whether the imported `members` from this Import become public or private `members` of the `importOwningPackage`.

**Operations**

importedMembership() : Membership [0..*]

Returns the Memberships of the `importedPackage` whose `memberElements` are to become imported `members` of the `importOwningPackage`. By default, this is the set of publicly visible Memberships of the `importedPackage`, but this may be overridden in specializations of Import

**Constraints**

No constraints.

#### 7.2.4.3.3 Membership

**Description**

Membership is a Relationship between a Package and an Element that indicates the Element is a `member` of (i.e., is contained in) the Package. The Membership may define a `memberName` for the Element as a member of the Package and specifies whether or not the Element is publicly visible as a `member` of the Package from outside the Package. The Element may be owned by the Membership, in which case it is an `ownedMember` of the Package, or it may be referenced but not owned, in which case it is effectively individually imported into the Package.

**General Classes**

Relationship

**Attributes**

memberElement : Element {redefines target}

The Element that becomes a `member` of the `membershipOwningPackage` due to this Membership.

memberName : String [0..1]

The name of the `memberElement` in the namespace defined by the `membershipOwningPackage`.

membershipOwningPackage : Package {subsets package, redefines source, owningRelatedElement}

The Package of which the `memberElement` becomes a `member` due to this Membership.

ownedMemberElement : Element [0..1] {subsets memberElement, ownedRelatedElement}

The `memberElement` of this Membership if it is owned by the Membership as an `ownedRelatedElement`.

visibility : VisibilityKind

Whether or not the Membership of the memberElement in the membershipOwningPackage is publicly visible outside that Package. Unless the membershipOwningPackage is a Type, visibility must be either public or private.

**Operations**

isDistinguishableFrom(other : Membership) : Boolean

Whether this Membership is distinguishable from a given other Membership. By default, this is true if the memberName of this Membership is either empty or is different the memberName of the other Membership, or if the metaclass of the memberElement of this Membership is different than the metaclass of the memberElement of the other Membership. But this may be overridden in specializations of Membership.

**Constraints**

No constraints.

### 7.2.4.3.4 Package

**Description**

A Package is an Element that contains other Elements, known as its members, via Membership Relationships with those Elements. Some of the members of a Package may be owned by the Package. The rest are imported into the Package, either as unowned memberElements of owned Memberships of the Package or via Import Relationships with other Packages.

A Package also acts as a namespace for its members that consists of the memberNames specified by all the Memberships in the Package. If a Membership specifies a memberName, then that is the name of the corresponding memberElement relative to the namespace defined by the Package. Note that the same Element may be the memberElement of multiple Memberships in a Package (though it may be owned at most once), each of which may define a separate alias for the Element relative to the Package namespace.

**General Classes**

Element

**Attributes**

/importedMembership : Membership [0..*] {subsets membership, ordered}

The Memberships in this Package that result from Import Relationships between the Package and other Packages. This excludes any Membership from one imported Package that would be indistinguishable from a Membership imported from another Package or from an ownedMembership of this Package.

/member : Element [0..*] {ordered}

The set of all member Elements of a Package, derived as the memberElements of all memberships of the Package.

/membership : Membership [0..*] {ordered, union}

All Memberships in this Package, defined as the union of ownedMemberships and importedMemberships.

ownedImport : Import [0..*] {subsets sourceRelationship, ownedRelationship, ordered}

The Import Relationships for which this Package is the `importingPackage`.

/ownedMember : Element [0..*] {subsets member, ordered}

The owned `members` of this Package, derived as the `ownedMemberElements` of the `ownedMemberships` of the Package.

ownedMembership : Membership [0..*] {subsets membership, sourceRelationship, ownedRelationship, ordered}

The Memberships for which this Package is the `membershipOwningPackage`.

**Operations**

excludeCollisions(mem : Membership [0..*]) : Membership [0..*]

Exclude from the given set `mem` of Memberships those that would not be distinguishable from each other if imported into this Package.

namesOf(element : Element) : String [0..*]

Return the names of the given `element` as it is known in the namespace defined by this Package.

publicMemberships() : Membership [0..*]

Return the publicly visible Memberships of this Package, which includes those `ownedMemberships` that are with a `visibility` of `public` and those `importedMemberships` that where imported via Import Relationships with a `visibility` of `public`.

**Constraints**

packageMembers

[no documentation]

```
member = membership.memberElement
```

packageOwnedMembers

[no documentation]

```
ownedMember = ownedMembership.ownedMemberElement
```

packageDistinguishibility

[no documentation]

```
membership->forAll(m1 | membership->forAll(m2 | m1 <> m2 implies m1.isDistinguishableFrom(
```

packageImportedMembership

[no documentation]

```
importedMembership = excludeCollisions(ownedImport.importedMembership())->select(m1 | owne
```

#### 7.2.4.3.5 VisibilityKind

**Description**

VisibilityKind is an enumeration whose literals specify the visibility of a Membership of an Element in a Package outside of that Package. Note that "visibility" specifically restricts whether an Element in a Package may be referenced by name from outside the Package and only otherwise restricts access to an Element as provided by specific constraints in the abstract syntax (e.g., preventing the import or inheritance of private Elements).

**General Classes**

No general classes.

**Literal Values**

package

Only valid if the owning Package of a Membership is a Type. Indicates that the Membership is visible to all Elements within the nearest enclosing Package that is not a Type to which it would have been visible if it had public visibility, but that it is not visible outside the nearest owning Package that is not a Type (or if there is no such Package).

private

Indicates a Membership is not visible outside its owning Package.

protected

Only valid for if the owning Package of a Membership is a Type. Indicates that the Membership is visible outside its owning Type only if inherited by direct or indirect specializations of the Type.

public

Indicates that a Membership is publicly visible outside its owning Package.

# 7.3 Core

## 7.3.1 Core Overview

### 7.3.1.1 General

The Core layer specializes the Root layer to add the minimum modeling constructs for specifying systems as they are build or operate (that have semantics). *Semantics* is about alignment of models and the things being modeled (real, simulated, or imagined things of any kind, including objects, links between them, and performances of behaviors). Models give conditions for how things should be (a specification of things), or for a model to be an accurate reflection of things (an explanation or record of things). See discussion in 6.1.

KerML specifies the alignment above by *classification.* Things being modeled are aligned with models when the model has elements that classify those things. Core introduces Type, the most general kind of model element that classifies things (real or simulated) when used in models. Classifiers are Types that classify things, such as cars, people, and processes being carried out, as well as how they are related by Features, including chains of relationships (for "nested" Features). Features are Types that classify just the (chains of) relationships. Classifiers include how things are related to enable them to be identified by those relationships.  For example, cars owned by people who live a particular city might be required to be registered. These cars are identified by a chain of two relationships, first ownership of the car, then the residence of the owner.

Taxonomies are supported by Generalizations between Types (Superclassing for Classifiers, Subsetting and Redefinition for Features). Specialized Types classify all the things their more general Types do (via one or more Generalizations). This means things classified by a specialized Type have all the Features (via `features`) of its general Types (sometimes referred to as "inheriting" features from general to specific Types). FeatureTyping (the kinds of "values" a feature might have) is Generalization between a Feature and another Type.

The syntax and semantics for Types, Classifiers, and Features (see 7.3.3, 7.3.2, and 7.3.4, respectively) are described informally in their Overview subclauses, and then formally in their Concrete Syntax, Abstract Syntax, and Semantics subclauses.The mathematical term *universe* is used in the Overview subclauses, which is the set of all things potentially being modeled, separately from how they are related (see 7.3.1.2).

#### 7.3.1.2 Mathematical Preliminaries

The following are model theoretic terms, explained in terms of this specification:

- *Vocabulary*: Model elements conforming to abstract syntax and additional restrictions given in this subclause.
- *Universe*: All (real or virtual) things the vocabulary could possibly be about.
- *Interpretation*: The relationship between vocabulary and mathematical structures made of elements of the universe.

The *semantics* of KerML are restrictions on the interpretation relationship, given in this subclause and the Semantics subclauses. This subclause also defines the above terms for KerML. They are used by the mathematical semantics in the rest of the specification.

A vocabulary $V = ( V_T, V_C, V_F )$ is a 3-tuple where:

- $V_T$ is a set of types (model elements classified by Type or its specializations, see 7.3.2.3).
- $V_C \subseteq V_T$ is a set of classifiers (model elements classified by Classifier or its specializations, see 7.3.3.3), including at least *Base::Anything* from KerML model library, see 8.2).
- $V_F \subseteq V_T$ is a set of features (model elements classified by Feature or its specializations, see 7.3.4.3), including at least `Base::things` from the KerML model library (see 8.2).
- $V_T = V_C \cup V_F$

An interpretation $I = ( \Delta, \cdot^T )$ for $V$ is a 2-tuple where:

- $\Delta$ is a non-empty set (*universe*), and
- $\cdot^T$ is an (*interpretation*) function relating elements of the vocabulary to sets of sequences of elements of the universe. It has domain $V_T$ and co-domain that is the power set of $S$, where

  $$S = \cup_{i \in Z^+} \Delta^i$$

  $S$ is the set of all n-ary Cartesian products of $\Delta$ with itself, including 1-products, but not 0-products, which are called *sequences*. The Semantics subclauses give other restrictions on the interpretation function.

The phrase *result of interpreting* a model (vocabulary) element refers to sequences paired with the element by $\cdot^T$. This specification also refers to this as the *interpretation* of the model element, for short.

The function $\cdot^{minT}$ specializes $\cdot^T$ to the subset of sequences in an interpretation that have no others as tails, except when applied to *Anything*

$$\boxed{\forall t \in \text{Type}, s_1 \in S \;\; s_1 \in (t)^{minT} \equiv s_1 \in (t)^T \land (t \neq \text{Anything} \Rightarrow (\forall s_2 \in S \;\; s_2 \in (t)^T \land s_2 \neq s_1 \Rightarrow \neg \mathit{tail}(s_2, s_1)))}$$

The following functions, adapted from [DOL], Appendix F.4.1 (Semantic Conformance of UML With DOL, Preliminaries), operate on sequences:

- *length* is a synonym for DOL's *sequence-length*.

$$\forall s \;\; length(s) \equiv \text{form:sequence-length}(s)$$

- *head* is true if the first sequence is the same as the second for some of it, starting at the beginnings of both, otherwise is false.

$$\forall s_1, s_2 \;\; head(s_1, s_2) \Rightarrow \text{form:Sequence}(s_1) \wedge \text{form:Sequence}(s_2)$$
$$\forall s_1, s_2 \;\; head(s_1, s_2) \equiv (length(s1) \leq length(s_2)) \wedge$$
$$(\forall p, x \;\; p \in Z^+ \wedge p \geq 1 \wedge p \leq length(s_1) \Rightarrow$$
$$\text{form:in-position-count}(s_1, p, x) = \text{form:in-position-count}(s_2, p, x))$$

- *tail* is true if the first sequence is the same as the second for some of it, ending at the ends of both, otherwise is false:

$$\forall s_1, s_2 \; tail(s_1, s_2) \Rightarrow \text{form:Sequence}(s_1) \wedge \text{form:Sequence}(s_2)$$
$$\forall s_1, s_2 \; tail(s_1, s_2) \equiv (length(s1) \leq length(s_2)) \wedge$$
$$(\forall p, x(p \in Z^+) \wedge (p > length(s_2) - length(s_1)) \wedge (p \leq length(s_2) \Rightarrow$$
$$\text{form:in-position-count}(s_1, p, x) = \text{form:in-position-count}(s_2, p, x)))$$

- *concat* is true if the first sequence has the second as head, the third as tail, and its length is the sum of the lengths of the other two, otherwise is false:

$$\forall s_0, s_1, s_2 \;\; concat(s_0, s_1, s_2) \Rightarrow \text{form:Sequence}(s0) \wedge \text{form:Sequence}(s1) \wedge \text{form:Sequence}(s2)$$
$$\forall s_0, s_1, s_2 \;\; concat(s_0, s_1, s_2) \equiv (length(s_0) = length(s_1) + length(s_2)) \wedge head(s_1, s_0) \wedge tail(s_2, s_0)$$

## 7.3.2 Types

### 7.3.2.1 Types Overview

#### Types

Type is the most general kind of model element in KerML that has semantics (in the sense of 6.1 and 7.3.1.2). Types classify things in the modeled universe and/or (chains of) relationships between those things (see 7.3.1.1). The set of things and (chains of) relationships classified by a Type is the *extent* of the Type, each member of which is an *instance* of the Type. Everything in the modeled universe and all (chains of) relationships between them are instances of the Type *Anything* in the Base model library (see 8.2).

**Note.** Referring to things and (chains of) relationships between them collectively as instances is for clarity of explanation only. The mathematical semantics treats both as sequences (see 7.3.1.2 and the Semantics subclauses).

Types give conditions for what things must be in their extent and what must not be (*sufficient* and necessary conditions, respectively). The simplest conditions directly identify instances that must be in or not in the extent. Other conditions can give characteristics of instances indicating they must be in or not in the extent. For example, a type Car could require every instance in its extent (everything it classifies) to have four wheels, which means anything that does not have four wheels is not in its extent (necessary condition). It does not mean all four wheeled things are in the extent (are cars), however (necessary conditions are usually stated as what must be true of all instances in the extent, even though they only determine what is not). Alternatively, Car could require all four wheeled things to be in its extent (sufficient condition).

Conditions in KerML are always necessary and can be indicated as sufficient for all conditions of a Type as needed, whereupon the sufficient conditions are the negation of the necessary ones. For example, if Car requires all instances to be four wheeled (necessary), and then is also is indicated as sufficient, its extent will include all four wheeled things and no others. The original (necessary) condition excludes everything not four wheeled, then indicating Car is sufficient brings in all four wheeled things. These conditions apply to all procedures that determine the extent of Types, including logical solvers, inference engines, and software.

A Type can also be *abstract,* which means that all instances of the Type must also be instances of at least one (possibly indirect)specialization of it (which must not be abstract, that is, must be *concrete*), see Generalization below.

### Generalization

Generalizations are Relationships between a `specific` Type and a `general` one, indicating that all instances of the `specific` Type are instances of the `general` one (the extent of the `specific` Type is a subset of the `extent` of the `general` one, which might be the same set). This means instances of the `specific` Type have all the features of the `general` ones, referred to syntactically as *inheriting* features from general to specific Types, see below. It also enables Generalization Relationships to form cycles, which means all Types in the cycle have the same instances (same extent).

### Classifiers and Features

Types divide into Classifiers and Features (7.3.3 and 7.3.4, respectively). Classifiers classify things in the universe and how they are related, while Features classify only how they are related (see 7.3.4.1). Types must be Classifiers or Features, but not both. However, they can generalize each other in models. Classifiers generalizing Features limit what things the Features can relate to, see FeatureTyping in 7.3.4.1. Classifiers generalized by Features can have no instances, because Classifiers must include things in the modeled universe, regardless of how they are related, whereas Features cannot include those.

**Note.**Types as the union of Classifiers and Features is required by the mathematical semantics (see 7.3.1.2), but not by the abstract syntax. This specification does not give semantics to Types that are not Classifiers or Features.

### Membership in Types

Types are Packages, enabling them to have members via Membership Relationships to other Elements identified as their `memberships` (see 7.2.4). These include `inheritedMemberships`, which are certain Memberships from the `general` Types of their `ownedGeneralizations`. The `memberNames` of all `inheritedMemberships` must be distinct from each other and from the `memberNames` of all `ownedMemberships`. A Membership that would otherwise be imported is also hidden by an `inheritedMembership` of with the same memberName, just as in the case of an ownedMembership (see 7.2.4.1).

Except for name conflicts, as described above, the `inheritedMemberships` include all visible and protected Memberships of the general Types. *Protected* Memberships are all owned and inherited Memberships of the general Type whose visibility is the VisibilityKind `protected` (imported Memberships can never have `protected` visibility). This means `protected` Memberships are Memberships that are only visible to their owning Type and to (direct or indirect) specializations of it.

**Note.** Name conflicts due to inherited Memberships can be resolved by redefining them to give non-conflicting `memberNames` (see 7.3.4).

### Feature Membership

Features are members of Types via FeatureMembership Relationships, a kind of Membership, identified as the `features` of a Type (the inverse of this for Feature is `featuringTypes`), see 7.3.4.

### Multiplicity

The number of instances in the extent of a Type (*cardinality*) is constrained by its *multiplicity*. A Multiplicity is a Feature whose values are natural numbers that are the only ones allowed for the cardinality of its featuringType (every Multiplicity is the feature of exactly one Type). A Type can have at most one feature that is a Multiplicity, identified as its multiplicity. Cardinality for Classifiers is the number of things in the modeled universe classified by the it. For Features that are not an end Feature, cardinality is the number of values of the Feature for a specific instance of its featuringTypes.

**Note.** See 7.4.10 in Kernel for specifying numeric ranges for multiplicities, rather than each number separately as above.

An EndFeatureMembership is a FeatureMembership that specifies the memberFeature as an *end* Feature of the owningType of the EndFeatureMembership. EndFeatureMembership has exactly the same meaning as FeatureMembership, except that the semantics of Multiplicity is different for end Features. End Features are used primarily in the definition of Associations and Connectors (see 7.4.3 and 7.4.4, respectively).

### Conjugation

Conjugation is a Relationship between two Types in which the conjugatedType inherits visible and protected Memberships from the originalType, except the direction of input and output Features is reversed. FeatureMemberships with direction in relative to the originalType are treated as having direction of out relative to the conjugatedType, and vice versa for direction in treated as out. FeatureMemberships with with no direction or direction inout in the originalType are inherited without change. Types can participate as conjugatedTypes in at most one Conjugation Relationship, and they shall not also be the specific Type in any Generalization relationship.

### 7.3.2.2 Concrete Syntax

#### 7.3.2.2.1 Types

```
Type (m : Membership) : Type =
    ( isAbstract ?= 'abstract' )? 'type'
    TypeDeclaration(this, m) TypeBody(this)

TypeDeclaration (t : Type, m : Membership) =
    (t.isSufficient ?= 'all' )? Identification(t, m)
    ( t.ownedFeatureMembership += MultiplicityMember )?
    ( SpecializationPart(t) | ConjugationPart(t) )+

SpecializationPart (t : Type) =
      SPECIALIZES t.ownedGeneralization += OwnedGeneralization
      ( ',' t.ownedGeneralization += OwnedGeneralization )*

ConjugationPart (t : Type) =
      CONJUGATES t.ownedConjugator += OwnedConjugation

MultiplicityMember : FeatureMembership =
    ownedMemberFeature = Multiplicity

TypeBody (t : Type) =
    ';' | '{' TypeBodyElement(t)* '}'

TypeBodyElement (t : Type) : Type =
      t.documentation += OwnedDocumentation
    | t.ownedMembership += NonFeatureTypeMember(t)
    | t.ownedFeatureMembership += FeatureTypeMember
    | t.ownedImport += PackageImport

NonFeatureTypeMember (t : Type) : Membership =
    TypeMemberPrefix(this) NonFeatureElement(this, t)

FeatureTypeMember : FeatureMembership =
    FeatureMember | EndFeatureMember

TypeMemberPrefix (m : Membership) =
    ( m.ownedRelationship += PrefixAnnotation )*
    ( m.visibility = VisibilityIndicator )?

VisibilityIndicator : VisibilityKind =
    PackageVisibilityIndicator | 'protected'
```

Similarly to the generic Package notation (see 7.2.4.2), the representation of a Type includes a *declaration* and a *body.*

A Type is declared using the keyword **`type`**, optionally followed by a `nameId` and/or `name`. In addition, a Type declaration defines either one or more `ownedGeneralizations` for the Type (for notation, see 7.3.2.2.2) or a `conjugator` for the Type (for notation, see 7.3.2.2.3).

A Type is specified as abstract (`isAbstract = true`) by placing the keyword **`abstract`** before the keyword **`type`**. A Type is specified as sufficient (`isSufficient = true`) by placing the keyword **`all`** after the keyword **`type`**. (This notational placement of the **`abstract`** and **`all`** keywords is also consistent in the notation for Classifiers and Features.)

```
abstract type A specializes Base::Anything;
type all x specializes A, Base::things;
```

The multiplicity of a Type is specified after any identification of the Type, between square brackets `[...]` (see also 7.4.10 on MultiplicityRanges).

```
// This Type has exactly one instance.
type Singleton[1] specializes Base::Anything;
```

The body of a Type is specified as for a generic Package, by listing the members between curly braces `{...}` (see 7.2.4.2). However, unlike non-Type Packages, Types can have protected members. A protected member is indicated using the keyword **`protected`**, instead of **`public`** or **`private`**. In addition, Features that are declared as `ownedMembers` of a Type are automatically considered to be `ownedFeatures` of the Type, related by FeatureMemberships (see 7.3.2.2.4).

```
type Super specializes Base::Anything {
    private package P {
        type Sub specializes Super;
    }
    protected feature f : P::Sub;
}
```

#### 7.3.2.2.2 Generalization

```
Generalization (m : Membership) : Generalization =
    ( 'generalization' Identification(this, m) )?
    'subtype' specific = [QualifiedName]
    SPECIALIZES general = [QualifiedName] ';'

OwnedGeneralization : Generalization =
    general = [QualifiedName]
```

A Generalization Relationship is declared using the keyword **`generalization`**, optionally followed by a `humanId` and/or a `name`. The qualified name of the `specific` Type is then given after the keyword **`subtype`**, followed by the qualified name of the `general` Type after the keyword **`specializes`**. The symbol **`:>`** can be used interchangeably with the keyword **`specializes`**.

```
generalization Gen subtype A specializes B;
generalization subtype x :> Base::things;
```

If no humanId or name is given, then the keyword **`generalization`** may be omitted.

```
subtype C specializes A;
subtype C specializes B;
```

An `ownedGeneralization` of a Type is defined as part of the declaration of the Type, rather than in a separate declaration, by including the qualified names of the general Type in a list after the keyword **`specializes`** (or the symbol `:>`).

```
type C specializes A, B;
type f :> Base::things;
```

#### 7.3.2.2.3 Conjugation

```
Conjugation (m : Membership) =
    'conjugation' Identification(this, m)
    'type' conjugatedType = [QualifiedName]
    CONJUGATES originalType = [QualifiedName] ';'

OwnedConjugation : Conjugation =
    originalType = [QualifiedName]
```

A Conjugation Relationship is declared using the keyword **`conjugation`**, followed by a `humanId` and/or a `name`. The qualified name of the `conjugatedType` is then given after the keyword **`type`**, followed by the qualified name of the `originalType` after the keyword **`conjugates`**. The symbol ~ can be used interchangeably with the keyword **`conjugates`**.

```
type Original specializes Base::Anything {
    in feature Input;
}
type Conjugate1 specializes Base::Anything;
type Conjugate2 specializes Base::Anything;
conjugation c1 type Conjugate1 conjugates Original;
conjugation c2 type Conjugate2 ~ Original;
```

An `ownedConjugator` for a Type is defined as part of the declaration of the Type, rather than in a separate declaration, by including the qualified name of the originalType after the keyword **`conjugates`** (or the symbol ~).

```
type Conjugate1 conjugates Original;
type Conjugate2 ~ Conjugate1;
```

A Type can be the `conjugatedType` of at most one Conjugation Relationship. A `conjugatedType` shall not have any `ownedGeneralizations`.

### 7.3.2.2.4 Feature Membership

```
FeatureMember : FeatureMembership =
    TypeMemberPrefix(this) FeatureMemberFlags(this)
    ( ownedMemberFeature = FeatureElement(this)
    | 'feature'? ( memberName = NAME )?
      'is' memberFeature = [QualifiedName] ';'
    )

EndFeatureMember : EndFeatureMembership =
    TypeMemberPrefix(this) 'end' FeatureMemberFlags(this)
    ownedMemberFeature = FeatureElement(this)

FeatureMemberFlags (m : FeatureMembership) =
    ( m.direction = FeatureDirection )?
    ( m.isComposite ?= 'composite' | m.isPortion ?= 'portion' )?
    ( m.isReadOnly ?= 'readonly' )? ( m.isDerived ?= 'derived' )?
    ( m.isPort ?= 'port' )?

FeatureDirection : FeatureDirectionKind =
    'in' | 'out' | 'inout'
```

The body of a Type contains declarations of the Elements that are the members of of the Type, just as in the generic notation for a Package (see 7.2.4). However, unlike a non-Type Package, a Type can be the featuringType of those of its members that are Features. The features of a Type are declared in two ways:

- A Feature declared directly in the body of a Type automatically becomes an `ownedFeature` of that Type (see 7.3.4.2).
- A non-owned feature of a Type is declared using the same feature keyword used to declare an ownedFeature, but with the qualified name of the Feature given after the keyword **is**. Such a declaration may also include a `memberName` for the Feature relative to the `featuringType`. If no explicit `memberName` is given, then the `name` of the Feature (if any) is used as the implicit default.

As kinds of Types, the above also applies to the bodies of Classifiers (see 7.3.3) and Features (see 7.3.4.2). A Feature may also be an imported into a Type like an other Element (see 7.2.3), in which case it is related to the importing Type by a regular Membership, not a FeatureMembership, and, so, does not become one of the `features` of the Type.

```
feature person : Person;
classifier Person {
    feature age : ScalarValues::Integer; // This Feature member is owned.
    feature parent[2] is person;         // This Feature member is not owned.
    import person as personAlias;        // This is not a FeatureMembership.
}
```

Whether a Feature member is owned or not, there are a number of additional properties of its FeatureMembership that can be flagged by adding specific keywords to its declaration. If present these are always specified in the following order, after any visibility indicator:

1. **in**, **out**, **inout** – Specifies that the FeatureMembership has the indicated direction.

2. **`composite`** or **`portion`** – Specifies either `isComposite` = `true` or `isPortion` = `true` (specifying both is not allowed).
3. **`readonly`** – Specifies `isReadOnly` = `true`.
4. **`derived`** – Specifies `isDerived` = `true`.
5. **`port`** – Specifies `isPort` = `true`.

**Implementation Note.** As of 2020-09, the notation for **`readonly`** and **`derived`** have not been implemented yet.

```
classifier Fuel {
    portion feature fuelPortion : Fuel;
}
classifier Tank {
    port feature fillPort {
        in feature fuelFlow: Fuel;
    }
    composite feature fuel : Fuel;
}
```

A Feature can also be declared to be an `endFeature` (that is, related to the Type by an EndFeatureMembership) using the keyword **end**, which is placed before the above flag keywords (if any). Any kind of Type can have `endFeatures`, but they are mostly used in Associations (see 7.4.3) and Connectors (see 7.4.4).

```
assoc VehicleRegistration {
    end feature owner[1] : Person;
    end feature vehicle[*] : Vehicle;
}
```

#### 7.3.2.3 Abstract Syntax

### 7.3.2.3.1 Overview

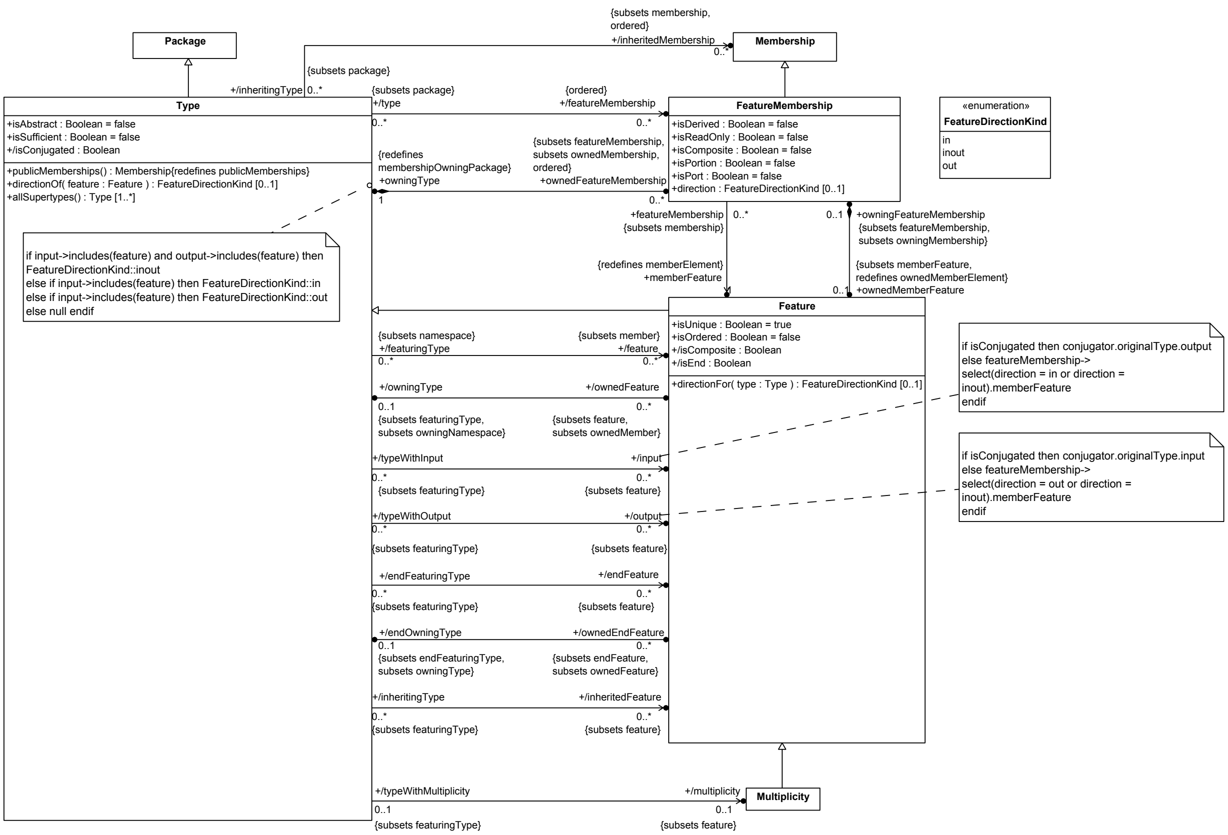


**Figure 11. Types**

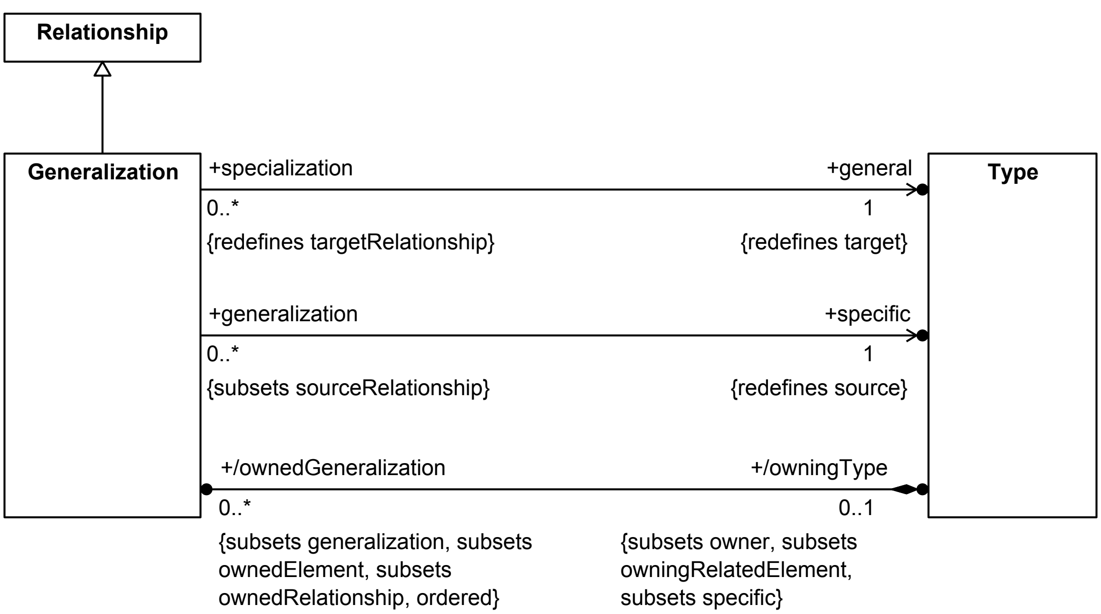


**Figure 12. Generalization**

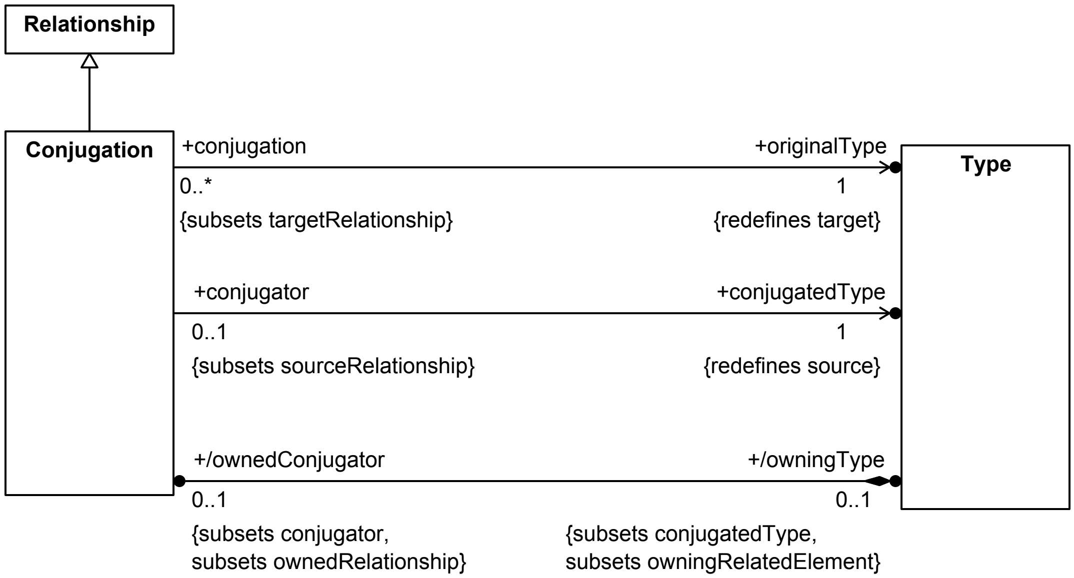


**Figure 13. Conjugation**

### 7.3.2.3.2 Conjugation

**Description**

Conjugation is a Relationship between two types in which the `conjugatedType` inherits all the Features of the `originalType`, but with all `input` and `output` Features reversed. That is, any Features with a FeatureMembership with `direction` *in* relative to the `originalType` are considered to have an effective `direction` of *out* relative to the `conjugatedType` and, similarly, Features with `direction` *out* in the `originalType` are considered to have an effective `direction` of *in* in the `originalType`. Features with `direction` *inout*, or with no `direction`, in the `originalType`, are inherited without change.

A Type may participate as a `conjugatedType` in at most one Conjugation relationship, and such a Type may not also be the `specific` Type in any Generalization relationship.

**General Classes**

Relationship

**Attributes**

conjugatedType : Type {redefines source}

The Type that is the result of applying Conjugation to the `originalType`.

originalType : Type {redefines target}

The Type to be conjugated.

/owningType : Type [0..1] {subsets conjugatedType, owningRelatedElement}

The `conjugatingType` of this Type that is also its `owningRelatedElement`.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.3.2.3.3 EndFeatureMembership

**Description**

An EndFeatureMembership is a FeatureMembership that specifies the `memberFeature` as an `endFeature` of the `owningType` of the EndFeatureMembership. EndFeatureMembership has exactly the same meaning as FeatureMembership, except that the semantics of Multiplicity is different for `endFeatures`.

An `endFeature` is always considered to map each domain entity to a single co-domain entity, whether or not a Multiplicity is given for it. If a Multiplicity is given for an `endFeature`, rather than giving the co-domain cardinality for the Feature as usual, it specifies a cardinality constraint for *navigating* across the `endFeatures` of the `featuringType` of the end Feature. That is, if a Type has *n* `endFeatures`, then the Multiplicity of any one of those end Features constrains the cardinality of the set of values of that Feature when the values of the other *n-1* end Features are held fixed.

**General Classes**

FeatureMembership

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.3.2.3.4 FeatureDirectionKind

**Description**

FeatureDirectionKind enumerates the possible kinds of `direction` that a Feature may be given as a member of a Type.

**General Classes**

No general classes.

**Literal Values**

in

Values of the Feature on each instance of its domain are determined externally to that instance and used internally.

inout

Values of the Feature on each instance are determined either as *in* or *out* directions, or both.

out

Values of the Feature on each instance of its domain are determined internally to that instance and used externally.

#### 7.3.2.3.5 FeatureMembership

**Description**

FeatureMembership is a Membership for a Feature in a Type that also asserts that all instances of the domain of the Feature are instances of the `owningType`.

**General Classes**

Membership

**Attributes**

direction : FeatureDirectionKind [0..1]

Determines how values of the Feature are determined or used, see FeatureDirectionKind.

isComposite : Boolean

Whether the values of the Feature can exist after the instance of the domain no longer does.

isDerived : Boolean

Whether the values of the Feature can always be computed from the values of other Features.

isPort : Boolean

Whether the Feature is visible externally to instances of the Feature's domain.

isPortion : Boolean

Whether the values of the Feature are contained in the space and time of instances of the Feature's domain.

isReadOnly : Boolean

Whether the values of the Feature can change over the lifetime of an instance of the domain.

memberFeature : Feature {redefines memberElement}

The Feature that this FeatureMembership relates to its `owningType`, making it a `feature` of the `owningType`.

ownedMemberFeature : Feature [0..1] {subsets memberFeature, redefines ownedMemberElement}

A `memberFeature` that is owned by this FeatureMembership and hence an `ownedFeature` of the `owningType`.

owningType : Type {redefines membershipOwningPackage}

The Type that owns this FeatureMembership.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.3.2.3.6 Generalization

**Description**

Generalization is a Relationship between two Types that requires all instances of the `specific` type to also be instances of the `general` Type (i.e., the set of instances of the `specific` Type is a *subset* of those of the `general` Type, which might be the same set).

**General Classes**

Relationship

**Attributes**

general : Type {redefines target}

A Type with a superset of all instances of the `specific` Type, which might be the same set.

/owningType : Type [0..1] {subsets specific, owner, owningRelatedElement}

The Type that is the `specific` Type of this Generalization and owns it as its `owningRelatedElement`.

specific : Type {redefines source}

A Type with a subset of all instances of the `general` Type, which might be the same set.

**Operations**

No operations.

**Constraints**

generalizationSpecificNotConjugated

The `specific` Type of a Generalization cannot be a conjugated Type.

```
not specific.isConjugated
```

#### 7.3.2.3.7 Type

**Description**

A Type is a Package that is the most general kind of Element supporting the semantics of classification. A Type may be a Classifier or a Feature, defining conditions on what is classified by the Type (see also the description of `isSufficient`).

**General Classes**

Package

**Attributes**

/endFeature : Feature [0..*] {subsets feature}

All `features` related to this Type by EndFeatureMemberships.

/feature : Feature [0..*] {subsets member}

The `memberFeatures` of the `featureMemberships` of this Type.

/featureMembership : FeatureMembership [0..*] {ordered}

All FeatureMemberships that have the Type as source. Each FeatureMembership identifies a `feature` of the Type.

/inheritedFeature : Feature [0..*] {subsets feature}

All the `memberFeatures` of the `inheritedMemberships` of this Type.

/inheritedMembership : Membership [0..*] {subsets membership, ordered}

All Memberships inherited by this Type via Generalization or Conjugation.

/input : Feature [0..*] {subsets feature}

All `features` identified by FeatureMemberships that have a `direction` of `in` or `inout`. (See FeatureDirectionKind.)

isAbstract : Boolean

Indicates whether instances of this Type must also be instances of at least one of its specialized Types.

/isConjugated : Boolean

Indicates whether this Type has an `ownedConjugator`. (See Conjugation.)

isSufficient : Boolean

Whether all things that meet the classification conditions of this Type must be classified by the Type.

(A Type gives conditions that must be met by whatever it classifies, but when `isSufficient` is false, things may meet those conditions but still not be classified by the Type. For example, a Type *`Car`* that is not sufficient could require everything it classifies to have four wheels, but not all four wheeled things would need to be cars. However, if the type *`Car`* were sufficient, it would classify all four-wheeled things.)

/multiplicity : Multiplicity [0..1] {subsets feature}

The one `feature` (at most) of this Type that is a Multiplicity, which constrains the cardinality of the Type. This Multiplicity must redefine the `multiplicity` (if it has one) of any `general` Type of any `Generalization` of this Type.

/output : Feature [0..*] {subsets feature}

All `features` identified by FeatureMemberships that have a `direction` of `out` or `inout`. (See FeatureDirectionKind.)

/ownedConjugator : Conjugation [0..1] {subsets ownedRelationship, conjugator}

A Conjugation owned by this Type for which the Type is the `originalType`.

/ownedEndFeature : Feature [0..*] {subsets endFeature, ownedFeature}

All `endFeatures` of this Type that are `ownedFeatures`.

/ownedFeature : Feature [0..*] {subsets feature, ownedMember}

The `ownedMemberFeatures` of the `ownedFeatureMemberships` of this Type.

ownedFeatureMembership : FeatureMembership [0..*] {subsets ownedMembership, featureMembership, ordered}

All FeatureMemberships that have the Type as source and are owned by it. Each FeatureMembership identifies a Feature of the Type.

/ownedGeneralization : Generalization [0..*] {subsets generalization, ownedElement, ownedRelationship, ordered}

The Generalizations owned by this Type for which the Type is the `specific` Type.

**Operations**

allSupertypes() : Type [1..*]

```
post:  result = let g : Bag = generalization.general in
   g->union(g->collect(allSupertypes()))->flatten()->asSet()->including(self)
```

directionOf(feature : Feature) : FeatureDirectionKind [0..1]

If the given feature is a feature of this type, then return its direction relative to this type, taking conjugation into account.

publicMemberships() : Membership

**Constraints**

typeOwnedGeneralizations

[no documentation]

```
ownedGeneralization = generalization->intersection(ownedElement)
```

typeMultiplicity

The `multiplicity` of this Type is all its `features` that are Multiplicities. (There must be at most one.)

```
multiplicity = feature->select(oclIsKindOf(Multiplicity))
```

#### 7.3.2.4 Semantics

##### Required Generalizations to Model Library

See Required Relationships to Model Library in 7.3.3.4 .

##### Type Semantics

The interpretation of Types in a model shall satisfy the following rules:

1. All sequences in the interpretation of a Type are in the interpretations of its generalizing Types.

$$\forall t_g, t_s \in V_T \;\; t_g \in t_s.\text{generalization.general} \Rightarrow (t_s)^T \subseteq (t_g)^T$$

### 7.3.3 Classifiers

#### 7.3.3.1 Classifiers Overview

##### Classifiers

Classifiers are Types that classify things in the modeled universe, regardless of how Features relate them, as well how they are related by Features (7.3.4.1). (See Classifiers and Features in 7.3.2.1 about how they are related.)

##### Superclassing

Superclassing is a kind of Generalization that restricts its `specific` and `general` Types to be Classifiers, identifying them as `subclass` and `superclass`, respectively. The `subclass` can omit any instances of the extent of the `superclass` that would be inconsistent with the semantics of Classifier (see 7.3.3.4).

#### 7.3.3.2 Concrete Syntax

##### 7.3.3.2.1 Classifiers

```
Classifier (m : Membership) : Classifier =
    ( isAbstract ?= 'abstract' ) 'classifier'
    ClassifierDeclaration(this, m) TypeBody(this)

ClassifierDeclaration (t : Type, m : Membership)
    ( t.isSufficient ?= 'all' )? Identification(t, m)
    ( t.ownedFeatureMembership += MultiplicityMember )?
    ( SuperclassingPart(t) | ConjugationPart(t) )?

SuperclassingPart (t : Type) =
      SPECIALIZES t.ownedSuperclassing += OwnedSuperclassing
      ( ',' t.ownedSuperclassing += OwnedSuperclassing )*
```

The notation for a Classifier is the same as the generic notation for a Type, except using the keyword **`classifier`** rather than **`type`**. However, any general Types referenced in a **`specializes`** list must be Classifiers, and the Generalizations defined are specifically Superclassings. A Classifier is also not required to have any `ownedSuperclassings` explicitly specified. If no explicit Superclassing is given for a Classifier, and the Classifier is not conjugated, then the Classifier is given a default Superclassing to the most general base Classifier *Anything* from the *Base* model library (see 8.2).

```
classifier Person { // Default superclass is Base::Anything.
    feature age : ScalarValues::Integer;
}
classifier Child specializes Person;
```

The declaration of a Classifier may also specify that the Classifier is a `conjugatedType` (see 7.3.2.2), in which case the `originalType` must also be a Classifier.

```
classifier FuelInPort {
    in feature fuelFlow : Fuel;
}
classifier FuelOutPort conjugates FuelInPort;
```

#### 7.3.3.2.2 Superclassing

```
Superclassing (m : Membership) =
    ( 'generalization' Identification(this, m) )?
    'subclass' subclass = [QualifiedName]
    SPECIALIZES superclass = [QualifiedName] ';'

OwnedSuperclassing : Superclassing =
    superclass = [QualifiedName]
```

A Superclassing Relationship is declared using the keyword **`generalization`**, optionally followed by a `humanId` and/or a `name`. The qualified name of the `subclass` is then given after the keyword **`subclass`**, followed by the qualified name of the `superclass` after the keyword **`specializes`**. The symbol `:>` can be used interchangeably with the keyword **`specializes`**.

```
generalization Super subclass A specializes B;
generalization subclass B :> A;
```

If no `humanId` or `name` is given, then the keyword **`generalization`** may be omitted.

```
subclass C specializes A;
subclass C specializes B;
```

An `ownedSuperclassing` of a Classifier is defined as part of the declaration of the Classifier, rather than in a separate declaration, by including the qualified name of the `superclass` in a list after the keyword **`specializes`** (or the symbol `:>`).

```
classifier C specializes A, B;
```

### 7.3.3.3 Abstract Syntax

### 7.3.3.3.1 Overview

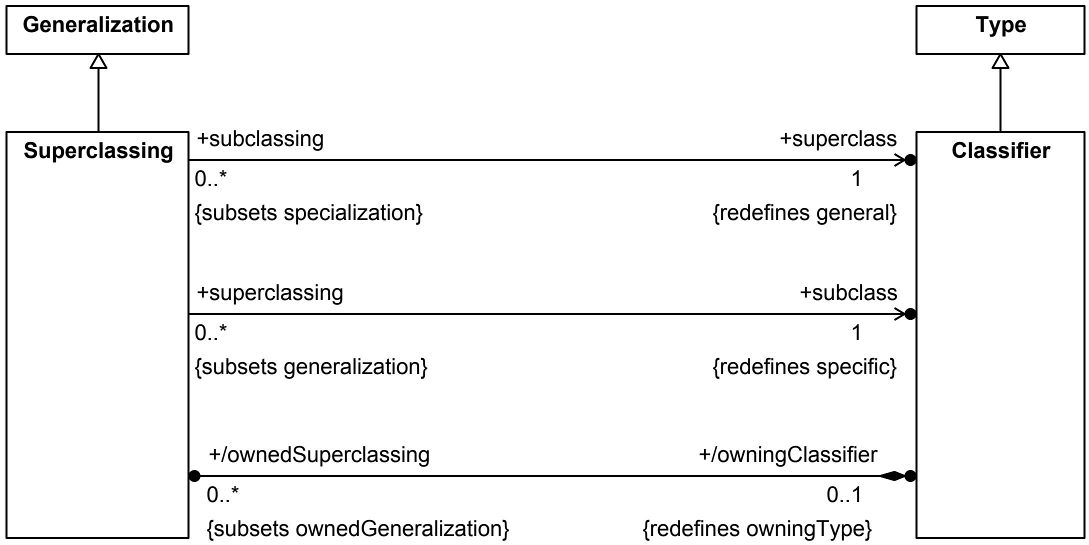


**Figure 14. Classifiers**

### 7.3.3.3.2 Classifier

**Description**

A Classifier is  a Type for model elements that classify:

- Things (in the universe) regardless of how Features relate them.  These are sequences of exactly one thing (sequence of length 1).
- How the above things are related by Features. These are sequences of multiple things (length > 1).

Classifiers that classify relationships (sequence length > 1) must also classify the things at the end of those sequences (sequence length =1).  Because of this, Classifiers specializing Features cannot classify anything (any sequences).

**General Classes**

Type

**Attributes**

/ownedSuperclassing : Superclassing [0..*] {subsets ownedGeneralization}

All Superclassing Relationships owned by this Classifier for which the Classifier is the `subclass`.

**Operations**

No operations.

**Constraints**

classifierOwnedSuperclassings

[no documentation]

```
ownedSuperclassing = ownedGeneralization->intersection(superclassing)
```

classifierSpecializesAnything

[no documentation]

```
allSupertypes()->includes(Kernel Library::Anything)
```

#### 7.3.3.3.3 Superclassing

**Description**

Superclassing is Generalization in which both the `specific` and `general` Types are Classifiers. This means all instances of the specific Classifier are also instances of the general Classifier.

**General Classes**

Generalization

**Attributes**

/owningClassifier : Classifier [0..1] {redefines owningType}

The more specific Classifier in the pair linked by Superclassing and that owns the relationship.

subclass : Classifier {redefines specific}

The more specific Classifier in the pair linked by Superclassing.

superclass : Classifier {redefines general}

The more general Classifier in the pair linked by Superclassing.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.3.3.4 Semantics

#### Required Generalizations to Model Library

1. All Types shall directly or indirectly specialize *Base::Anything* (see 8.2.2.1), including Classifiers (implied by Rule 2 below combined with the definition of $.^{T}$ in 7.3.1.2).

#### Classifier Semantics

The interpretation of the Classifiers in a model shall satisfy the following rules:

1. If the interpretation of a Classifier includes a sequence, it also includes the 1-tail of that sequence.

$\forall c \in V_C, s_1 \in S \;\; s_1 \in (c)^T \Rightarrow (\forall s_2 \in S \;\; tail(s_2, s_1) \wedge length(s_2) = 1 \Rightarrow s_2 \in (c)^T)$

2. The interpretation of the Classifier *Anything* includes all sequences of all elements of the universe.

$(\text{Anything})^T = S$

## 7.3.4 Features

### 7.3.4.1 Features Overview

#### Features

A Feature is a Type that classifies how things in the modeled universe are related, including by chains of relationships. Relations between things can themselves be treated as things, allowing relations between relations (recurring as many times as needed). A Feature relates instances in the intersection of the extents of its `featuringTypes` (the *domain*) with instances in the intersection of the extents of its `types` (the *co-domain*). The domain of Features with no `featuringTypes` is the Type *Anything* from the Base model library (see 7.3.2.1 and 8.2). (See Classifiers and Features in 7.3.2.1 about how they are related.)

#### Feature Typing

FeatureTyping is a kind of Generalization that restricts its `specific` Type to be a Feature, identifying it as `typedFeature`, while its `general` Type is not restricted, but identified by `type` (which must be a Classifier or another Feature, see Classifiers and Features in 7.3.2.1). FeatureTyping can form cycles of Features to mean the extents of Features are the same, like any Generalization, but a Classifier in the cycle will make it unsatsifiable (see Generalization, and Classifiers and Features, in 7.3.2.1).

The `ownedTypings` of a Feature are those FeatureTypes for which the Feature is a `typedFeature` and which are owned by the Feature. The `types` of a Feature are the union of the `types` of all its `ownedTypings` with all the `types` of the `subsettedFeatures` of the Feature (see Subsetting below), excluding any Types that directly or indirectly generalize any others.

#### Subsetting

Subsetting is a kind of Generalization that restricts its `specific` and `general` Types to be Features, identifying them as `subsettingFeature` and `subsettedFeature`, respectively. Any aspect of the `subsettingFeature` can be restricted relative to `subsettedFeature` as, such as the (co)domain and multiplicity (see below).

#### Redefinition

Redefinition is a kind of Subsetting that requires the things identified by (values of) the `redefinedFeature` and the `redefiningFeature` (specialized from `subsettingFeature` and `subsettedFeature`, respectively) to be the same on each instance (separately) of the domain of the `redefiningFeature`. This means any restrictions on the values of `redefiningFeature` relative to `redefinedFeature`, such as on the (co)domain or multiplicity, also apply to the values of `redefinedFeature` (on each instance of the domain of the `redefiningFeature`), and vice versa.

Redefinition also requires the `owningType` of the `redefiningFeature` to (indirectly) specialize the `owningType` (or Anything) of the `redefinedFeature` (redefining Type), and to *not* inherit the `redefinedFeature` into its namespace. This enables the `redefiningFeature` to have the same name as the `redefinedFeature` if desired. However, the absence of the `redefiningFeature` from namespace of the redefining Type does not prevent it from having values on instances of that Type, see above.

### Mathematical Semantics

Types are interpreted as sequences of one or more things from the modeled universe, where each thing in the sequence is related to the next by a Feature. Classifier interpretations include sequences of length 1, as well as longer sequences ending in the things in their 1-sequences ("navigations" to those things). These longer sequences are interpretations of Features. Feature sequences can be divided in two, beginning with an interpretation of its domain, and ending with an interpretation of its co-domain (its *value*). In the simplest case, a Feature as exactly one `featuringType` and exactly one `type`, both of which are Classifiers. The interpretations of such a Feature are pairs (sequences of length 2) of a thing from a 1-sequence of the `featuringType` and a thing from a 1-sequence of the `type`. Interpretations of the `type` Classifier includes the Feature pairs ("navigations" to the last thing in the sequence). This way of interpreting Classifiers enables FeatureTyping to be a kind of Generalization that restricts the Feature interpretations to sequences that end (lead to) 1-sequences of its `type`. Features can also have Features as their `featuringType` or `type`. ("nested" features). In this case, the sequences will be longer than 2.

### 7.3.4.2 Concrete Syntax

#### 7.3.4.2.1 Features

```
Feature (m : Membership) : Feature =
    ( isAbstract ?= 'abstract' )? 'feature'?
    FeatureDeclaration(this, m) ValuePart(this)?
    TypeBody(this)

FeatureDeclaration (f : Feature, m : Membership) =
    ( f.isSufficient ?= 'all' )? Identification(f, m)
    ( FeatureSpecializationPart(f) | ConjugationPart(f) )?

FeatureSpecializationPart(f : Feature) =
      FeatureSpecializationPart(f)+ MultiplicityPart(f)? FeatureSpecialization(f)*
    | MultiplicityPart(f) FeatureSpecialization(f)*

MultiplicityPart (f : Feature) =
    f.ownedFeatureMembership += MultiplicityMember
    ( f.isOrdered ?= 'ordered' ( !f.isUnique ?= 'nonunique' )?
    | !f.isUnique ?= 'nonunique' ( isOrdered ?= 'ordered' )? )?

FeatureSpecialization (f : Feature) =
    Typings(f) | Subsettings(f) | Redefinitions(f)

Typings (f : Feature) =
      TypedBy(f) ( ',' f.ownedTyping += OwnedFeatureTyping )*

TypedBy (f : Feature) =
    TYPED_BY f.ownedTyping += OwnedFeatureTyping

Subsettings (f : Feature) =
    Subsets(f) ( ',' f.ownedSubsetting += OwnedSubsetting )*

Subsets (f : Feature) =
    SUBSETS f.ownedSubsetting += OwnedSubsetting

Redefinitions (f : Feature) =
    Redefines(f) ( ',' f.ownedRedefinition += OwnedRedefinition )*

Redefines (f : Feature) =
    REDEFINES ownedRelationship += OwnedRedefinition
```

The notation for a Feature is similar to the generic notation for a Type, except using the keyword **`feature`** rather than **`type`**. Further, a Feature can have any of three kinds of Generalization: FeatureTyping (see 7.3.4.2.2), Subsetting (see 7.3.4.2.3) and Redefinition (see 7.3.4.2.4). In general, clauses for the different kinds of Generalization can appear in any order in a Feature declaration.

```
feature x typed by A, B subsets f redefines g;

// Equivalent declaration:
feature x redefines g typed by A subsets f typed by B;
```

If no Subsetting (or Redefinition) is explicitly specified for a Feature, and the Feature is not conjugated, then the Feature is given a default Subsetting of the most general base Feature *things* from the *Base* model library (see 8.2). This is true even if a FeatureTyping is given for the Feature.

```
abstract feature person : Person; // Default subsets Base::things.
feature child subsets person;
```

The declaration of a Feature may also specify that the Feature is a `conjugatedType` (see 7.3.2.2.3), in which case the `originalType` must also be a Feature.

```
classifier Tanks {
    port feature fuelInPort {
        in feature fuelFlow : Fuel;
    }
    port feature fuelOutPort ~ fuelInPort;
}
```

As for any Type, the `multiplicity` of a Feature can be given in square brackets `[...]` after any identification of the Feature. However, the `multiplicity` for a Feature can also be placed *after* one or more initial Generalization clauses in the Feature declaration. In particular, this allows a notation style for multiplicity consistent with that used in previous modeling languages. It is also useful when redefining a Feature without giving an explicit name (see 7.3.4.2.4).

```
feature parent[2] : Person;
feature mother : Person[1] :> parent;
```

In addition, if an explicit `multiplicity` is given for a Feature, then that can be followed by either or both of the following keywords (in either order):

- **`nonunique`** – Specifies `isUnique = false` (the default is true).
- **`ordered`** – Specifies `isOrdered = true`.

#### 7.3.4.2.2 Feature Typing

```
FeatureTyping (m : Membership) : FeatureTyping =
    'generalization' Identification(this, m)
    'typing' typedFeature = [QualifiedName]
    TYPE_OF type = [QualifiedName] ';'

OwnedFeatureTyping : FeatureTyping =
    type = [QualifiedName]
```

A FeatureTyping Relationship is declared using the keyword **`generalization`**, optionally followed by a `humanId` and/or a `name`. The qualified name of the `typedFeature` is then given after the keyword **`typing`**, followed by the qualified name of the `type` after the keyword **`typed by`**. The symbol **`:`** can be used interchangeably with the keyword **`typed by`**.

```
generalization t1 typing f typed by B;
generalization t2 typing g : A;
```

An `ownedTyping` is defined as part of the declaration of the Classifier, rather than in a separate declaration, by including the qualified name of the `type` in a list after the keyword **`typed by`** (or the symbol **`:`**).

```
feature f typed by A, B;
```

#### 7.3.4.2.3 Subsetting

```
Subsetting (m : Membership) : Subsetting =
    ( 'generalization' Identification(this, m) )?
    'subset' subsettingFeature = [QualifiedName]
    SUBSETS subsettedFeature = [QualifiedName] ';'

OwnedSubsetting : Subsetting =
    subsettedFeature = [QualifiedName]
```

A Subsetting Relationship is declared using the keyword **`generalization`**, optionally followed by a `humanId` and/or a `name`. The qualified name of the `subsettingFeature` is then given after the keyword **`subset`**, followed by the qualified name of the `subsettedFeature` after the keyword **`subsets`**. The symbol **`:>`** can be used interchangeably with the keyword **`subsets`**.

```
generalization Sub subset parent subsets person;
generalization subset mother subsets parent;
```

If no `humanId` or `name` is given, then the keyword **`generalization`** may be omitted.

```
subset rearWheels subsets wheels;
subset rearWheels subsets driveWheels;
```

An `ownedSubsetting` of a Feature is defined as part of the declaration of the Feature, rather than in a separate declaration, by including the qualified name of the `subsettedFeature` in a list after the keyword **`subsets`** (or the symbol **`:>`**).

```
feature rearWheels subsets wheels, driveWheels;
```

If a `subsettedFeature` is ordered, then the `subsettingFeature` must also be ordered. If the `subsettedFeature` is unordered, then the `subsettingFeature` will be unordered by default, unless explicitly flagged as `ordered`.

```
feature anyWheels[*] : Wheels;
classifier Automobile {
    composite feature wheels[4] ordered subsets anyWheels;
    composite feature driveWheels[2] ordered subsets wheels; // Must be ordered.
}
```

If a `subsettedFeature` is unique, then the `subsettingFeature` must not be specified as non-unique. If the `subsettedFeature` is non-unique, then the subsettingFeature will still be unique by default, unless specifically flagged as `nonunique`.

```
feature urls[*] nonunique : URL;
classifier Server {
    feature accessibleURLs subsets urls; // Unique by default.
    feature visibleURLs subset accessibleURLs; // Cannot be nonunique.
}
```

#### 7.3.4.2.4 Redefinition

```
Redefinition (m : Membership) : Redefinition =
    ( 'generalization' Identification(this, m) )?
    'redefinition' redefiningFeature = [QualifiedName]
    REDEFINES redefinedFeature = [QualifiedName] ';'

OwnedRedefinition : Redefinition =
    redefinedFeature = [QualifiedName]
```

A Redefinition Relationship is declared using the keyword **`generalization`**, optionally followed by a `humanId` and/or a `name`. The qualified name of the `redefiningFeature` is then given after the keyword **`redefinition`**, followed by the qualified name of the `redefinedFeature` after the keyword **`redefines`**. The symbol `:>>` can be used interchangeably with the keyword **`redefines`**.

```
generalization Redef redefinition LegalRecord::guardian redefines parent;
generalization redefinition Vehicle::vin redefines RegisteredAsset::identifier;
```

If no `humanId` or `name` is given, then the keyword **`generalization`** may be omitted.

```
redefinition Vehicle::vin redefines RegisteredAsset::identifier;
redefinition Vehicle::vin redefines legalIdentification;
```

An `ownedRedefinition` of a Feature is defined as part of the declaration of the Feature, rather than in a separate declaration, by including the qualified name of the `redefinedFeature` in a list after the keyword **`redefines`** (or the symbol `:>`).

```
feature vin redefines RegisteredAsset::identifier, legalIdentification;
```

The resolution of the qualified names of `redefinedFeatures` given in a Feature declared in the body of a Type shall follow the following special rules:

1. Resolve the qualified name beginning with the public and protected `members` of the local namespace of the `general` Types from each Generalization of the `owningType`.
2. If exactly one resolution is found, and the resolving Element is a Feature, then that is the resolution of the name for the `redefinedFeature`. Otherwise there is no resolution.

Note that the local namespace of the `owningType` is *not* included in the name resolution for `redefinedFeatures` in this way. Since `redefinedFeatures` are not inherited, they would not be included in the local namespace of the owning Type and, therefore, could not be referenced by an unqualified name. Despite this, the above special rules allow such a reference, because the name resolution begins with the namespaces of the general Types of the `owningType`, one of which must contain the `redefinedFeature`.

```
classifier RegisteredAsset {
    feature identifier : Identifier;
```

```
}
classifier Vehicle : RegisteredAsset { // Owning Type.
    feature vin redefines identifier;  // Legal even though "identifier" is not inher
}
```

If a `name` is not given in the declaration of a Feature with an `ownedRedefinition`, then, rather than the Feature having no `name`, it is implicitly given the same `name` as that of the `redefiningFeature` of its first `ownedRedefinition`, if any (which may itself be an implicit name, if the `redefinedFeature` is itself a `redefiningFeature`). (This is useful for constraining a `redefinedFeature`, while maintaining the same naming.)

```
classifier WheeledVehicle {
    composite feature wheels[1..*] : Wheel;
}
classifier MotorizedVehicle specializes WheeledVehicle {
    composite feature redefines wheels[2..4];
}
classifier Automobile specializes MotorizedVehicle {
    composite feature redefines wheels[4] : AutomobileWheel;
}
```

The restrictions on the specification of the ordering and uniqueness of a `subsettingFeature` (see 7.3.4.2.3) also apply to a `redefiningFeature`.

### 7.3.4.3 Abstract Syntax

#### 7.3.4.3.1 Overview

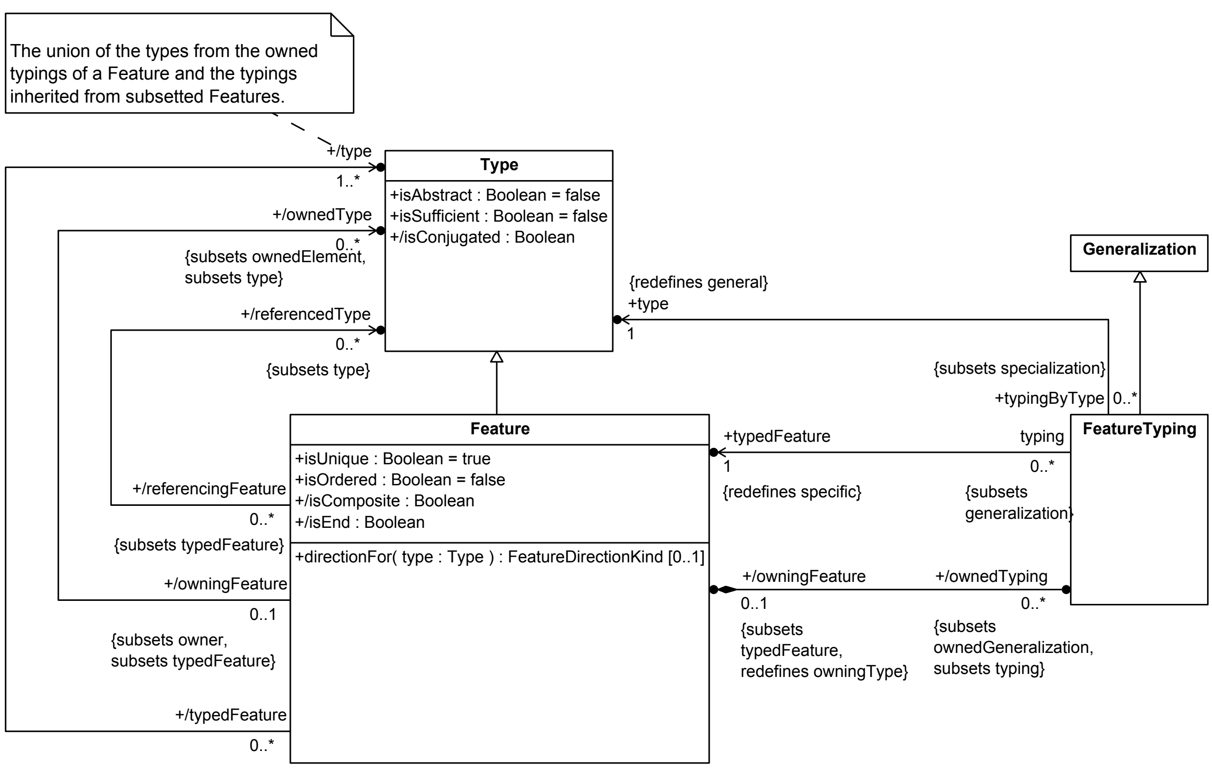


**Figure 15. Features**

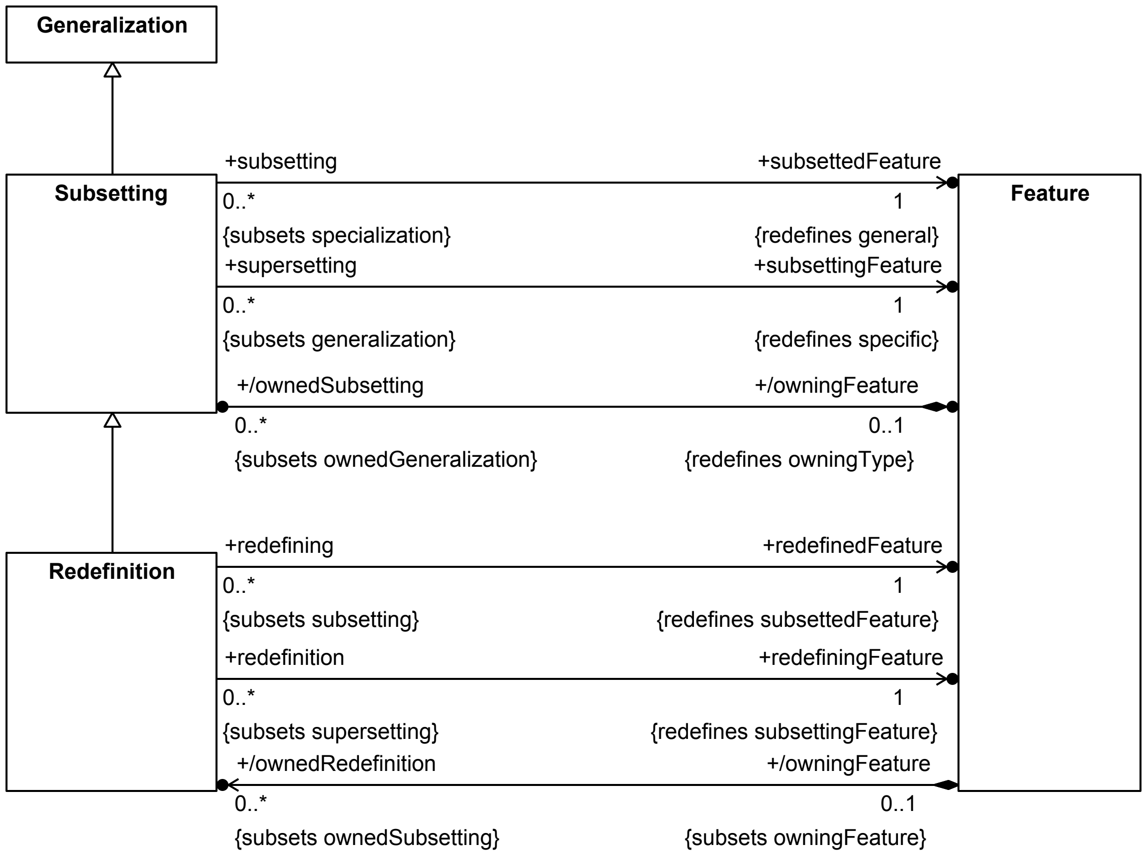


**Figure 16. Subsetting**

### 7.3.4.3.2 Feature

**Description**

A Feature is a Type that classifies sequences of multiple things (in the universe). These must concatenate a sequence drawn from the intersection of the Feature's `featuringTypes` (*domain*) with a sequence drawn from the intersection of its `types` (*co-domain*, range), treating (co)domains as sets of sequences. The domain of Features that do not have any `featuringTypes` is the same as if it were Anything. A Feature's `types` include at least Anything, which can be narrowed to other Classifiers by Redefinition.

In the simplest cases, a Feature's `featuringTypes` and `types` are Classifiers, its sequences being pairs (length = 2), with the first element drawn from the Feature's domain and the second element from its co-domain (the Feature "value"). Examples include cars paired with wheels, people paired with other people, and cars paired with numbers representing the car length.

Since Features are Types, their `featuringTypes` and `types` can be Features. When both are, Features classify sequences of at least four elements (length > 3), otherwise at least three (length > 2). The `featuringTypes` of *nested* Features are Features.

**General Classes**

Type

**Attributes**

/endOwningType : Type [0..1] {subsets endFeaturingType, owningType}

The Type that is related to this Feature by an EndFeatureMembership in which the Feature is an `ownedMemberFeature`.

/isComposite : Boolean

Whether the Feature is a composite `feature` of its `featuringType`, as given by whether `isComposite` is true for its owningFeatureMembership (see also FeatureMembership).

/isEnd : Boolean

Whether or not the `owningFeatureMembership` is an EndFeatureMembership, requiring a different interpretation of the `multiplicity` of the Feature. (See also EndFeatureMembership.)

isOrdered : Boolean

Whether an order exists for the values of this Feature or not.

isUnique : Boolean

Whether or not values for this Feature must have no duplicates or not.

/ownedRedefinition : Redefinition [0..*] {subsets ownedSubsetting}

The Redefinition Relationships owned by this Feature for which it is the `redefiningFeature`.

/ownedSubsetting : Subsetting [0..*] {subsets ownedGeneralization}

The Subsetting Relationships owned by this Feature for which it is the `subsettingFeature`.

/ownedType : Type [0..*] {subsets type, ownedElement}

The `types` of this Feature that are also owned by it.

/ownedTyping : FeatureTyping [0..*] {subsets ownedGeneralization, typing}

The FeatureTypings owned by this Feature for which it is the `typedFeature`.

owningFeatureMembership : FeatureMembership [0..1] {subsets owningMembership, featureMembership}

The FeatureMembership that owns this Feature as an `ownedMemberFeature`, determining its `owningType`.

/owningType : Type [0..1] {subsets featuringType, owningNamespace}

The Type that is the `owningType` of the `owningFeatureMembership` of this Type.

/referencedType : Type [0..*] {subsets type}

The `types` of this Feature that are not owned by it.

/type : Type [1..*]

The Types that restrict the values of this Feature, such that the values must by instances of all the `types`. The types of a Feature are derived from its `ownedFeatureTypings` and `ownedSubsettings`.

**Operations**

directionFor(type : Type) : FeatureDirectionKind [0..1]

Return the `directionOf` this Feature relative to the given `type`.

```
body: type.directionOf(self)
```

**Constraints**

featureOwnedTypes

[no documentation]

```
ownedType = type->intersection(ownedElement)
```

featureTypes

[no documentation]

```
type = typing.type
```

featureOwnedSubsettings

[no documentation]

```
ownedSubsetting = ownedGeneralization->intersection(subsetting)
```

featureIsEnd

[no documentation]

```
isEnd = owningFeatureMembership <> null and owningFeatureMembership.oclIsKindOf(EndFeature
```

featureIsComposite

[no documentation]

```
isComposite = owningFeatureMembership <> null and owningFeatureMembership.isComposite
```

featureOwnedRedefinitions

[no documentation]

```
ownedRedefinition = ownedSubsetting->intersection(redefining)
```

featureReferencedTypes

[no documentation]

```
referencedType = type - ownedElement
```

#### 7.3.4.3.3 FeatureTyping

**Description**

FeatureTyping is Generalization in which the `specific` Type is a Feature. This means the set of sequences of the (specific) `typedFeature` is a subset of the set of sequences of the (general) `type`. In the simplest case, the `type` is a Classifier, whereupon the `typedFeature` subset has sequences ending in things (in the modeled universe) in single-length sequences of the Classifier.

**General Classes**

Generalization

**Attributes**

/owningFeature : Feature [0..1] {subsets typedFeature, redefines owningType}

The Feature that owns this FeatureTyping, which must also be the `typedFeature`.

type : Type {redefines general}

The Type that is being applied by this FeatureTyping.

typedFeature : Feature {redefines specific}

The Feature that has its Type determined by this FeatureTyping.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.3.4.3.4 Multiplicity

**Description**

A Multiplicity is a Feature whose co-domain is a set of natural numbers that includes the number of sequences determined below, based on the kind of typeWithMultiplicity (featuringType):

- Classifiers: minimal sequences (the single length sequences of the Classifier).
- Features: sequences with the same feature-pair head. In the case of Features with Classifiers as domain and co-domain, these sequences are pairs, with the first element in a single-length sequence of the domain Classifier (head of the pair), and the number of pairs with the same first element being among the Multiplicity co-domain numbers.

Multiplicity co-domains (in models) can be specified by expressions that might vary in their results depending on the sequence of the `typeWithMultiplicity` on which the expression is evaluated.

**General Classes**

Feature

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.3.4.3.5 Redefinition

**Description**

Redefinition specializes Subsetting to require the `redefinedFeature` and the `redefiningFeature` to have the same values (on each instance of the domain of the `redefiningFeature`). This means any restrictions on the `redefiningFeature`, such as `type` or `multiplicity`, also apply to the `redefinedFeature` (on each instance of the `owningType` of the redefining Feature), and vice versa. The `redefinedFeature` might have values for instances of the `owningType` of the `redefiningFeature`, but only as instances of the `owningType` of the `redefinedFeature` that happen to also be instances of the `owningType` of the `redefiningFeature`. This is supported by the constraints inherited from Subsetting on the domains of the `redefiningFeature` and `redefinedFeature`. However, these constraints are narrowed for Redefinition to require the `owningTypes` of the `redefiningFeature` and `redefinedFeature` to be different and the `redefinedFeature` to not be imported into the `owningNamespace` of the `redefiningFeature`. This enables the `redefiningFeature` to have the same name as the `redefinedFeature` if desired.

**General Classes**

Subsetting

**Attributes**

redefinedFeature : Feature {redefines subsettedFeature}

The Feature that is redefined by the `redefiningFeature` of this Redefinition.

redefiningFeature : Feature {redefines subsettingFeature}

The Feature that is redefining the `redefinedFeature` of this Redefinition.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.3.4.3.6 Subsetting

**Description**

Subsetting is Generalization in which the `specific` and `general` Types that are Features. This means all values of the `subsettingFeature` (on instances of its domain, i.e., the intersection of its `featuringTypes`) are values of the `subsettedFeature` on instances of its domain. To support this, the domain of the `subsettingFeature` must be the same or specialize (at least indirectly) the domain of the `subsettedFeature` (via Generalization), and the range (intersection of a Feature's `types`) of the `subsettingFeature` must specialize the range of the `subsettedFeature`. The `subsettedFeature` is imported into the `owningNamespace` of the `subsettingFeature` (if it is not already in that namespace), requiring the names of the `subsettingFeature` and `subsettedFeature` to be different.

**General Classes**

Generalization

**Attributes**

/owningFeature : Feature [0..1] {redefines owningType}

The Feature that owns this Subsetting relationship, which must also be its `subsettingFeature`.

subsettedFeature : Feature {redefines general}

The Feature that is subsetted by the `subsettingFeature` of this Subsetting.

subsettingFeature : Feature {redefines specific}

The Feature that is a subset of the `subsettedFeature` of this Subsetting.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.3.4.4 Semantics

#### Required Generalizations to Model Library

1. All Features shall directly or indirectly specialize specialize `Base::things` (see 8.2.2.5) (implied by Rule 1 and 2 below combined with the definition of $.^T$ in 7.3.1.2).

#### Feature Semantics

The interpretation of the Features in a model shall satisfy the following rules:

1. The interpretations of features must have length greater than one.

$$\forall s \in S, f \in V_F \;\; s \in (f)^T \Rightarrow length(s) > 1$$

2. The interpretation of the Feature *things* is all sequences of length greater than one.

$$(\text{things})^T = \{\, s \mid s \in S \wedge length(s) > 1 \,\}$$

See other rules below.

Features interpreted as sequences of length two or more can be treated as if they were interpreted as sets of ordered pairs (binary relations), where the first and second elements of each pair are from the domain and co-domain of the Feature, respectively (see 7.3.4.1). The predicate *featurePair* below determines whether two sequences can be treated in this way.

Two sequences are a *feature pair* of a Feature if and only if the interpretation of the Feature includes a sequence $s_0$ such that following are true:

- The first sequence is a proper head of $s_0$ and is in the minimal interpretation of all `featuringTypes` of the Feature.
- The second sequence is a proper tail of $s_0$ and is in the minimal interpretations of all `types` of the Feature.

$$\begin{aligned}&\forall s_1, s_2 \in S, f \in V_F \;\; featurePair(s_1, s_2, f) \equiv \\ &\quad \exists s_0 \in S \;\; s_0 \in (f)^T \wedge concat(s_0, s_1, s_2) \;\wedge \\ &\qquad (\forall t_1 \in V_T \;\; t_1 \in f.\mathsf{featuringType} \Rightarrow s_1 \in (t_1)^{minT}) \;\wedge \\ &\qquad (\forall t_2 \in V_T \;\; t_2 \in f.\mathsf{type} \Rightarrow s_2 \in (t_2)^{minT})\end{aligned}$$

The interpretation of the Features in a model shall satisfy the following rules:

3. All sequences in an interpretation of a Feature are concatenations of sequences that are feature pairs of the Feature.

$$\forall s_0 \in S, f \in V_F \;\; s_0 \in (f)^T \Rightarrow \exists s_1, s_2 \in S \;\; concat(s_0, s_1, s_2) \wedge featurePair(s_1, s_2, f)$$

4. Values of `redefiningFeatures` are the same as the values of their redefinedFeatures restricted to the domain the `redefiningFeature`.

$$\begin{aligned}&\forall f_g, f_s \in V_F \;\; f_g \in f_s.\mathsf{redefinedFeature} \Rightarrow \\ &\quad (\forall s_1 \in S \;\; (\forall ft_s \in V_T \;\; ft_s \in f_s.\mathsf{featuringType} \Rightarrow s_1 \in (ft_s)^{minT}) \Rightarrow \\ &\qquad (\forall s_2 \in S \; (featurePair(s_1, s_2, f_s) \equiv featurePair(s_1, s_2, f_g))))\end{aligned}$$

5. The multiplicity of a Feature includes the cardinality of its values.

$$\forall s_1 \in S, f \in V_F \;\; \#\{s_2 \mid featurePair(s_1, s_2, f)\} \in (f.\mathrm{multiplicity})^T$$

# 7.4 Kernel

## 7.4.1 Kernel Overview

The Kernel layer completes the KerML metamodel. It specializes Core to add application-independent modeling capabilities beyond basic classification. These distinguish structure (things and limits on how they change over time) from processes (how things change over time). Structural elements include Classes and DataTypes (kinds of things), Associations between them, and Connectors (usages of Associations). Processing elements include Behaviors that coordinate other Behaviors via Steps (usages of Behaviors). Specialized processing elements include Functions, which are Behaviors that always yield a single result, and Expressions (usages of Functions), as well as Interactions, which combine Behaviors and Associations, and ItemFlows (Connectors using Interactions).

Kernel add semantics beyond the Core by specifying how model element reuse the Kernel Model Library (see Clause 8). These are specified as constraints in the metamodel. The simplest reuse is specialization (direct or indirect). For example, Classes must subclass Object from the *Objects* library model, while Features typed by Classes must subset `objects`. Similarly, Behaviors must subclass the *Performance* from the *Performances* library model, while Steps (Features typed by Behaviors) must subset `performances`. Sometimes more complicated reuse patterns are needed. For example, binary Associations (with exactly two ends) specialize *BinaryLink* from the library, and additionally require the ends of the Association redefine the `source` and `target` ends of *BinaryLink*.

The semantics of the library models are ultimately grounded in the Core semantics, so, specializing a base Type from an appropriate library model gives the required semantics to each syntactic category of elements in the Kernel. In the remainder of 7.4, these semantic requirements are formally described in the Semantics subclause for each of the abstract syntax packages in the Kernel layer.

The textual concrete syntax provides modelers a compact way to use the abstract syntax. It includes keywords that translate to patterns of using abstract syntax and libraries.The syntactic constructs in the Kernel act as "syntactic markers" for semantic patterns tying Kernel semantics to the Core. In this way, Kernel is a syntactic and semantic extension of the Core. It is an example of how other modeling languages can be built on KerML by further extending and specializing the Kernel metamodel.

## 7.4.2 Classification

### 7.4.2.1 Classification Overview

Classifiers in Kernel are divided into DataTypes, Classes, and Behaviors. DataTypes and Classes are specified in this subclause, while Behaviors are described in 7.4.5. This subclause begins to classify things in the modeled universe based on whether they are *distinguished* only by their relations to other things via Features (Datatypes) or can be distinguished without regard those relationships (Classes and Behaviors). This means DataTypes cannot also be Classes or Behaviors, or share instances with them.

#### Data Types

DataTypes are Classifiers that classify *DataValues* (see 8.2.2.2) and how they are related by Features to (other) things in the modeled universe. *DataValues* are distinguishable when they differ in how are they are related to other things. Since change over time means changing relationships, *DataValues* are outside of time (and space, compare to 8.3).

However, *DataValues* for some DataTypes are directly identified (*enumerated*), in which case they are distinguishable regardless of their relationship to other things. Such DataTypes include the *primitive types* defined in the Kernel Model Library *ScalarValues* package (see 8.10), and any subtypes of those.

#### Classes

Classes are Classifiers that classify *Objects* (see 8.4.2.5) and how they are related by Features to (other) things in the modeled universe. *Objects* are one kind of *Occurrence*, things that occupy time and space (see 8.3.2.5), the other being *Performances* (see 8.5). Relations between *Occurrences* can change over time, modeled as Features relating `timeSlices` of them, which are also *Occurrences*, or as *Links* existing between them for limited time (see 8.4.2.3). Portions of an *Occurrence* are classified the same as the *Occurrence* (or more specialized).

### 7.4.2.2 Concrete Syntax

#### 7.4.2.2.1 Data Types

```
DataType (m : Membership) : DataType =
    ( isAbstract ?= 'abstract' )? 'datatype'
    ClassifierDeclaration(this, m) TypeBody(this)
```

A DataType is declared as a Classifier (see 7.3.3.2.1), using the keyword **`datatype`**. If no `ownedSuperclassing` is explicitly given for the DataType, then it is implicitly given a default Superclassing to the DataType *DataValue* from the *Base* model library (see 8.2).

Either all of the `types` of a Feature shall be DataTypes, or none of the shall be. If they are all DataTypes, and no `ownedSubsetting` or `ownedRedefinition` is explicitly given in the Feature declaration, then the Feature is implicitly given a default Subsetting to the Feature *`dataValues`* from the *Base* model library (see 8.2).

```
datatype IdNumber specializes ScalarValues::Integer;
datatype Reading { // Subtypes Base::DataValue by default
    feature sensorId : IdNumber; // Subsets Base::dataValues by default.
    feature value : ScalarValues::Real;
}
```

#### 7.4.2.2.2 Classes

```
Class (m : Membership) : Class =
    ( isAbstract ?= 'abstract' )? 'class'
    ClassifierDeclaration(this, m) TypeBody(this)
```

A Class is declared as a Classifier (see 7.3.3.2.1), using the keyword **`class`**. If no `ownedSuperclassing` is explicitly given for the Class, then it is implicitly given a default Superclassing to the Class *Object* from the *Objects* model library (see 8.4).

Either all of the `types` of a Feature shall be Classes, or none of the shall be. If they are all Classes, and no `ownedSubsetting` or `ownedRedefinition` is explicitly given in the Feature declaration, then the Feature is implicitly given a default Superclassing to the Feature *`objects`* from the *Objects* model library (see 8.4), unless at least one of the types is an Association, in which case the default Superclassing shall be as specified in 7.4.3.2.

```
class Sensor { // Specializes Objects::Object by default.
    feature id : IdNumber;
    feature currentReading : ScalarValues::Real;
}
class SensorArray specializes Assembly {
    composite feature sensors[*] : Sensor; // Subsets Base::objects by default.
}
```

### 7.4.2.3 Abstract Syntax

#### 7.4.2.3.1 Overview

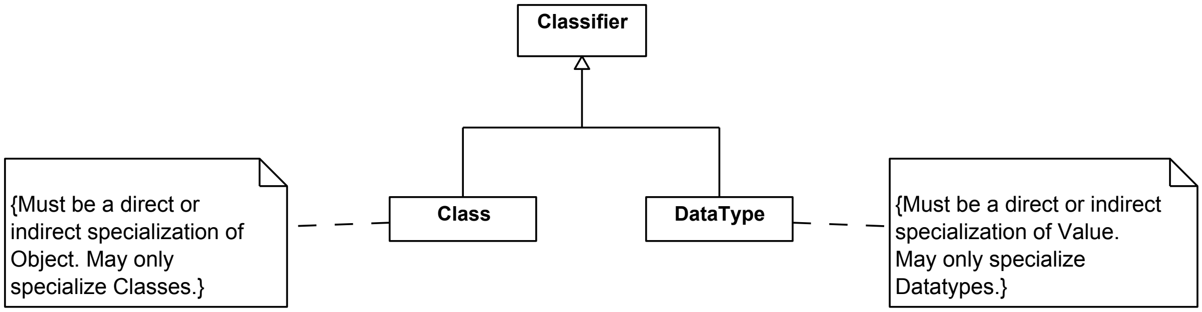


**Figure 17. Classification**

#### 7.4.2.3.2 Class

**Description**

A Class is a Classifier of things (in the universe) that can be distinguished without regard to how they are related to other things (via Features). This means multiple things classified by the same Class can be distinguished, even when they are related other things in exactly the same way.

**General Classes**

Classifier

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.2.3.3 DataType

**Description**

A DataType is a Classifier of things (in the universe) that can only be distinguished by how they are related to other things (via Features). This means multiple things classified by the same DataType

- Cannot be distinguished when they are related to other things in exactly the same way, even when they are intended to be about the same thing.
- Can be distinguished when they are related to other things in different ways, even when they are intended to be about the same thing.

**General Classes**

Classifier

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.2.4 Semantics

#### Required Generalizations to Model Library

1. DataTypes shall (indirectly) specialize *Base::DataValue* (see 8.2.2.2).
2. Features typed by DataTypes shall (indirectly) subset `Base::dataValues` (see 8.2.2.3).
3. Classes shall (indirectly) specialize *Objects::Object* (see 8.4.2.5).
4. Features typed by Classes shall (indirectly) subset `Objects::objects` (see 8.4.2.6).

#### DataType Semantics

For all the things at the end of sequences in the interpretation of a DataType, the heads of sequences ending in that thing shall be the same as heads of sequences ending in the other things.

#### Class Semantics

For all the things at the end of sequences in the interpretation of a Class (or Behavior), the heads of sequences ending in that thing shall be different than the heads of sequences ending in the other things.

### 7.4.3 Associations

#### 7.4.3.1 Associations Overview

Associations are Classes that classify *Links* (see 8.4.2.3) between things in the modeled universe, and how they are related by Features to those (other) things. At least two `ownedFeatures` of an Association must be `endFeatures` (see 7.3.2.1), its `associationEnds`. Associations with exactly two `associationEnds` classify BinaryLinks (see 8.4.2.1), and are called binary Associations.

An Association is also a Relationships between the `types` of its `associationEnds`, which are its `relatedTypes`. *Links* are between instances of an Association's `relatedTypes`.

The `features` of Associations that are not `endFeatures` characterize *Links* separately from the linked things. *Links* occupy time and space, like other *Objects*, with non-`endFeature` features having values that are potentially different things over time. However, the values of `endFeatures` do not change over time (though they can be *Occurrences* with Features whose values change over time).

### 7.4.3.2 Concrete Syntax

```
Association (m : Membership) : Association =
    ( isAbstract ?= 'abstract' )? 'assoc'
    ClassifierDeclaration(this, m) TypeBody(this)
```

An Association is declared a Classifier (see 7.3.3.2.1), using the keyword **assoc**. If no `ownedSuperclassing` is explicitly given for the Association, then it is implicitly given a default Superclassing to either the Association *BinaryLink* (if it is a binary Association) or the Class *Link* (otherwise), both of which are from the *Objects* library model (see 8.4).

If an Association has `ownedSuperclassings` whose `superclasses` are Associations, then these `superclass` Associations shall all have the same number of `associationEnds`. The `subclass` Association shall then have no more owned `associationEnds` than its `superclass` Associations. Each owned `associationEnd` of the `subclass` Association shall redefine an `associationEnd` of each of the `superclass` Associations. If no redefinition is given explicitly for an `associationEnd`, then it shall be considered to implicitly redefine the `associationEnd` at the same position, in order, of each `superClass` Association (including implicit defaults).

```
assoc Ownership { // Specializes Objects::BinaryLink by default.
    feature valuation : MonetaryValue;
    end feature owner[1..*] : LegalEntity; // Redefines BinaryLink::source.
    end feature ownedAsset[*] : Asset;     // Redefines BinaryLink::target.
}
assoc SoleOwnership specializes Ownership {
    end feature owner[1]; // Redefines Ownership::owner.
    // ownedAsset is inherited.
}
```

If an Association is not binary, then none of its `endFeatures` shall be composite. A binary Association shall have at most one composite `endFeature`.

```
assoc Assembly {
    end feature assembly[1] : Component;
    end composite feature parts[*] : Component;
}
```

If a Feature has one or more Associations as `types`, then these Associations shall all have the same number of `associationEnds`. If the Feature defines owned `endFeatures` in its body, then it shall have more than the number of `associationEnds` of its Association `types`. The owned `endFeatures` of such a Feature shall follow the same rules for redefinition of the `associationEnds` of its Association `types` as given above for the redefinition of the `associationEnds` of `superclass` Associations by a `subclass` Association.

If a Feature declaration has no explicit `ownedSubsettings` or `ownedRedefinitions`, and any of its `types` are binary Associations, then the Feature is implicitly given a default Subsetting to the Feature *`binaryLinks`* from the *Objects* model library (see 8.4). If some of the `types` are Associations, but not binary Associations, then it is given a default Subsetting to the Feature *`links`* from the *Objects* model library.

### 7.4.3.3 Abstract Syntax

#### 7.4.3.3.1 Overview

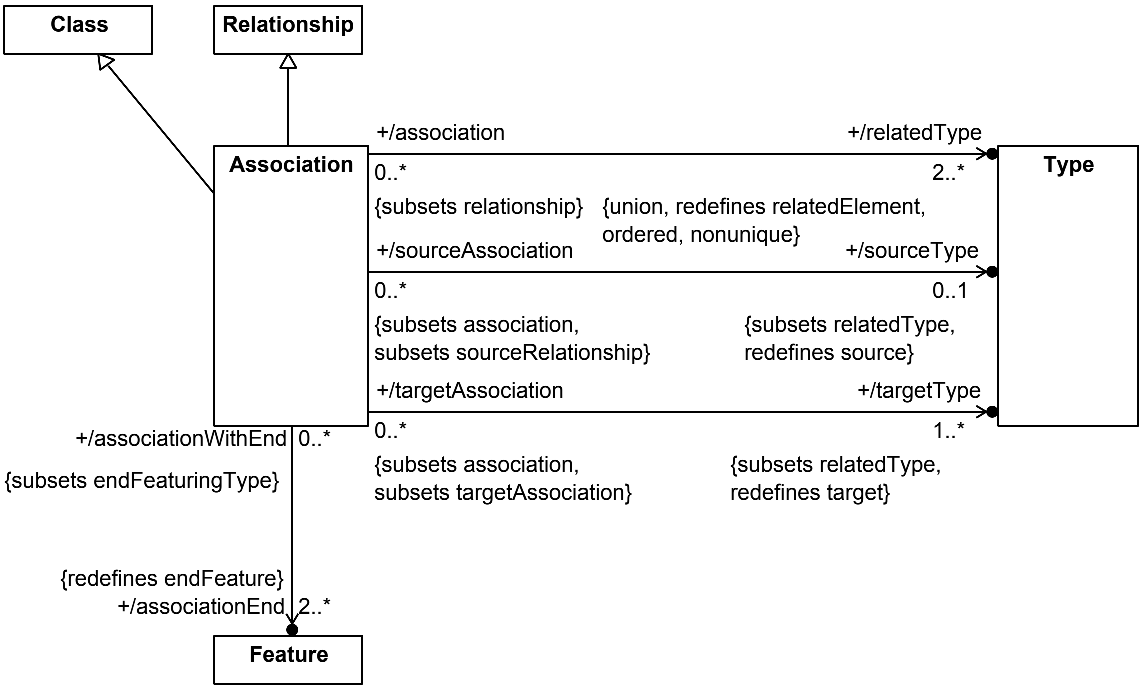


**Figure 18. Associations**

#### 7.4.3.3.2 Association

**Description**

An Association is a Relationship and a Class to enable classification of links between things (in the universe). The co-domains of the `associationEnd` Features are one of the `relatedTypes`, as co-domain and participants (linked things) of an Association identify each other.

**General Classes**

Relationship
Class

**Attributes**

/associationEnd : Feature [2..*] {redefines endFeature}

The `features` of the Association that that identifying the things that can be related by it. An Association must have at least two `associationEnds`. When it has exactly two, the Association is called a *binary* Association.

/owningConnector : Connector [0..1] {subsets owningFeature}

A Connector that owns the Association for which the Association is also a `type`.

/relatedType : Type [2..*] {redefines relatedElement, ordered, nonunique, union}

The `types` of the `endFeatures` of the Association, which are the `relatedElements` of the Association considered as a Relationship.

/sourceType : Type [0..1] {subsets relatedType, redefines source}

The source `relatedType` for this Association. If this is a binary Association, then the `sourceType` is the first `relatedType`, and the first `associationEnd` of the Association must redefine the `source` Feature of the Association *BinaryLink* from the Kernel Library. If this Association is not binary, then it has no `sourceType`.

/targetType : Type [1..*] {subsets relatedType, redefines target}

The target `relatedTypes` for this Association. This includes all the `relatedTypes` other than the `sourceType`. If this is a binary Association, then the `associationEnds` corresponding to the `relatedTypes` must all redefine the `target` Feature of the Association *BinaryLink* from the Kernel Library.

**Operations**

No operations.

**Constraints**

AssociationLink

[no documentation]

```
let numend : Natural = associationEnd->size() in
    allSupertypes()->includes(
        if numend = 2 then Kernel Library::BinaryLink
        else Kernel Library::Link)
```

associationRelatedTypes

[no documentation]

```
relatedTypes = associationEnd.type
```

### 7.4.3.4 Semantics

#### Required Generalizations to Model Library

1. Associations shall (indirectly) specialize *Objects::Link* (see 8.4.2.3).
2. Every `associationEnd` of an Association shall (indirectly) subset *`Link::participant`*.
3. Associations with exactly two `associationEnds` shall (indirectly) specialize *Objects::BinaryLink* (see 8.4.2.1).
4. Features typed by Associations shall (indirectly) specialize *`Objects::links`* (see 8.4.2.4).
5. Features typed by Associations with exactly two `associationEnds` shall (indirectly) specialize *`Objects::binaryLinks`* (see 8.4.2.2).

#### Association Semantics

Assocation `associationEnds` are given a special semantics compared to other `members`.

An N-ary Association of the form

```
assoc A {
    end feature e1;
    end feature e2;
```

```
        ...
        end feature eN;
    }
```

is semantically equivalent to the Core model

```
    class A specializes Objects::Link {
        end feature e1 subsets Objects::Link::participant;
        end feature e2 subsets Objects::Link::participant;
        ...
        end feature eN subsets Objects::Link::participant;
    }
```

The general semantics for the `multiplicity` of an `endFeature` is such that, even if a `multiplicity` other than `1..1` is specified, the Feature is required to effectively have multiplicity `1..1` relative to the *Link*. The *Link* instance for an Association is a tuple of `participants`, each one of which is a value of an `endFeature` of the Association. Note that the Feature *`Link::participant`* is declared **`readonly`**, meaning that the `participants` in a link cannot change once the link is created.

If an `associationEnd` has a `multiplicity` specified other than `1..1,` then this shall be interpreted as follows: For an Association with *N* `associationEnds`, consider the *i*-th `associationEnd` $e_i$. The multiplicity, ordering and uniqueness constraints specified for $e_i$ apply to each set of instances of the Association that have the same (singleton) values for each of the *N-1* `associationEnds` other than $e_i$.

For example, each instance of the Association

```
    assoc Ternary {
        end feature a[1];
        end feature b[0..2];
        end feature c[*] nonunique ordered;
    }
```

consists of three `participants`, one value for each of the `associationEnds` *`a`*, *`b`* and *`c`*. The multiplicities specified for the `associationEnds` then assert that:

1. For any specific values of *`b`* and *`c`*, there must be exactly one instance of *`Ternary`*, with the single value allowed for *`a`*.
2. For any specific values of *`a`* and *`c`*, there may be up to two instances of *`Ternary`*, all of which must have different values for *`b`* (default uniqueness).
3. For any specific values of *`a`* and *`b`*, there may be any number of instance of *`Ternary`*, which are ordered and allow repeated values for *`c`*.

**Submission Note.** The special semantics for the multiplicity of Features with EndFeatureMembership is under discussion. It will be finalized in a revised submission.

If an Association has an `ownedSuperclassing` to another Association, then its `associationEnds` redefine the `associationEnds` of the `superclass` Association. In this case, the `subclass` Association will indirectly specialize Link through a chain of Superclassings, and each of its `associationEnds` will indirectly subset `Links::participant` through a chain of redefinitions and a subsetting.

### Binary Association Semantics

Following the usual rules for the `associationEnds` of a specialized Association, the first `associationEnd` of the binary Association will redefine `BinaryLink::source` and the second `associationEnd` of the binary Association will redefine `BinaryLink::target`. The Association BinaryLink specializes *Link* and the Features `BinaryLink::source` and `BinaryLink::target` subset `Link::participant`. Therefore, the semantics for binary Associations are consistent with the semantics given above for Associations in general. In addition, the semantic model for a binary Association adds implicit nested *navigation* Features to each of the `associationEnds` of the Association, as described below.

A binary Association of the form

```
assoc A  {
    end feature e1;
    end feature e2;
}
```

is semantically equivalent to the Core model

```
class A specializes Objects::BinaryLink {
    end feature e1 redefines Objects::BinaryLink::source {
        feature e2 = A::e2(e1);
    }
    end feature e2 redefines Objects::BinaryLink::target {
        feature e1 = A::e2(e2);
    }
}
```

As shown above, the added navigation Feature for each end has the same name as the (effective) name of the *other* end. If the name of a navigation Feature is the same as an inheritable Feature from the `ownedGeneralizations` of the containing `associationEnd`, then the navigation Feature shall redefine that otherwise inherited Feature. The notation *`A::e2(e1)`* means "all values of the end `e2` of all instances of A that have the given value for the end e1". Therefore, for each value of *`A::e1`*, *`A::e1::e2`* gives the values of *`e2`* that have *`e1`* at the other end, that is, it defines a *navigation* across *`A`* from *`e1`* to *`e2`*. The meaning of *`A::e2::e1`* is similar.

**Submission Note.** The model for navigation across binary Associations is still under discussion and will be finalized in a revised submission.

## 7.4.4 Connectors

### 7.4.4.1 Connectors Overview

### Connectors

Connectors are Features that are typed by Associations (see 7.4.3), identifying (having values that are) Links (see 8.4.2.3). The values (Links) of a Connector are restricted to those that are

1. classified by the `types` of its `associationEnds`, regardless of the domain of the Connector.
2. identified by its `relatedFeatures` for each instance of the domain of the Connector.

Connectors could be called "instance-specific" associations (usages of associations), because their links are limited to be between things identified by instances of the Connector's domain (see below). For example, engines power wheels (an association), but only in cars (a connector), meaning the engine in a car powers the wheels in that same car, not the wheels in another car (as would be allowed with only the association).

Connectors are also Relationships between their `relatedFeatures`.

All Associations typing a Connector shall have the same number of `associationEnds`, which shall also be the number of owned `endFeatures` of the Connector, its `connectorEnds`. Each `connectorEnd` redefines one `associationEnd` from each `type` and subsets one of the `relatedFeatures` of the Connector. Connectors typed by binary Associations are called binary Connectors.

Instances identifying the things a Connector might link (see above) must be classified by a *context* Type, determined as follows:

1. Define a *feature membership path* to be a series of Features between a Package `P` and a Feature `f` such that:
   a. The first Feature in the series is an `ownedMember` of the Package `P`.
   b. Each successive Feature in the series is an `ownedFeature` of the previous Feature.
   c. The Feature `f` is an `ownedFeature` of the last Feature in the series.
2. Define the *relevant features* to be the Connector itself and eack of the `relatedFeatures` of the Connector, excluding `relatedFeatures` that are inherited `members` of the `owningType` of the Connector.
3. If there is a Type that begins feature membership paths to each of the relevant features, then this is the context Type.
4. If there are feature membership paths each of the relevant features that begin in a Package that is *not* a Type (though not necessarily the same Package for all of them), then the context Type is the library type *Base::Anything* (see 8.2.2.1).
5. Otherwise, the Connector has no context Type.

Connectors must have a context Type to identify *Links*.

### Binding Connectors

BindingConnectors are binary Connectors that require their `sourceFeature` and `targetFeature` to identify the same things (have the same values) on each instance of its domain. They are typed by SelfLink (which only links things in the modeled universe to themselves, see 8.4.2.7) and have end multiplicities of exactly 1. This requires a SelfLink to exist between each thing identified by the `sourceFeature` and one thing identified by `targetFeature`, and vice-versa.

Since the interpretations of DataTypes are disjoint from those of Classes and Behaviors (see 7.4.2 and 7.4.5), a Feature typed by DataTypes shall only be bound to another Feature typed by DataTypes. In the determination of the equivalence of such Features, indistinguishable data values shall be considered equivalent.

The binding of Features typed by Classes (or Behaviors) to another Feature typed by Classes (or Behaviors) indicates that the same objects (or performances) play the roles represented by each of the `relatedFeatures`.

BindingConnectors are used with FeatureValues (see 7.4.9).

### Successions

Successions are binary Connectors that require their `sourceFeature` and `targetFeature` to identify *Occurrences* that are ordered in time. They are typed by the Association *Objects::HappensBefore* from the model library (see 8.3.2.1), which links Occurrences that happen completely separately in time, with the Connector's `sourceFeature` being the *`earlierOccurrence`* and the `targetFeature` being the *`laterOccurrence`*.

### 7.4.4.2 Concrete Syntax

#### 7.4.4.2.1 Connectors

```
Connector (m : Membership) : Connector :
    ( isAbstract ?= 'abstract' )? 'connector'
    ConnectorDeclaration(this, m) TypeBody(this)

ConnectorDeclaration (c : Connector, m : Membership) : Connector =
    BinaryConnectorDeclaration(c, m) | NaryConnectorDeclaration(c, m)

BinaryConnectorDeclaration (c : Connector, m : Membership) : Connector =
    ( FeatureDeclaration(c, m)? 'from' | c.isSufficient ?= 'all' 'from'? )?
    c.ownedFeatureMembership += ConnectorEndMember 'to'
    c.ownedFeatureMembership += ConnectorEndMember

NaryConnectorDeclaration (c : Connector, m : Membership) : Connector =
    FeatureDeclaration(c, m)
    '(' c.ownedFeatureMembership += ConnectorEndMember ','
        c.ownedFeatureMembership += ConnectorEndMember
        ( ',' c.ownedFeatureMembership += ConnectorEndMember )* ')'

ConnectorEndMember : EndFeatureMembership :
    ( memberName = NAME '=>' )? ownedMemberFeature = ConnectorEnd

ConnectorEnd : Feature :
    ownedSubsetting += OwnedSubsetting
    ( ownedFeatureMembership += MultiplicityMember )?
```

A Connector is declared as a Feature (see 7.3.4.2) using the keyword **`connector`**. In addition, a Connector declaration includes a list of qualified names of the `relatedFeatures` of the Connector, between parentheses `(...)`, after the regular Feature declaration part and before the body of the Connector (if any). If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the Connector is implicitly given a default Subsetting to the Feature *`binaryLinks`* from the *Objects* model library (see 8.4), if it is a binary Connector, or to the Feature *`links`* from the *Objects* model library, if it is not a binary Connector. Note that, due to this default subsetting, if no `type` is explicitly given for a binary Connector, then it will implicitly have the type *BinaryLink*. However, *Link* is a Class, not an Association (because it has no `endFeatures`), so a `type` shall always be declared (directly or indirectly) for a non-binary Association.

```
assoc Mounting { // Specializes Objects::BinaryLink by default.
    end feature mountingAxle[1] : Axle;
    end feature mountedWheel[2] : Wheel;
}
class WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;

    // Subsets Objects::binaryLinks by default
    connector mount[2] : Mounting (axle, wheels);
}
```

By default, the `connectorEnds` of a Connecctor are declared in the same order as the `associationEnds` of the `types` of the Connector. However, if the Connector has a single `type`, then the `relatedFeatures` can be given in any order, with each `relatedFeature` paired with an `associationEnd` of the type using a notation of the form `e => f`, where `e` is the name of an `associationEnd` and `f` is the qualified name of a `relatedFeature`. In this case, the name of each `associationEnd` shall appear exactly once in the list of `connectorEnds` declarations.

```
class WheelAssembly {
    composite feature axle[0..1] : Axle;
    composite feature wheels[0..2] : Wheel;
    connector mount[2] : Mounting (
        mountedWheel => wheels,
        mountingAxle => axle);
}
```

A special notation can be used for a binary Connector, in which the source `relatedFeature` is referenced after the keyword **`from`**, and the target `relatedFeature` is referenced after the keyword **`to`**.

```
class WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;
    connector mount[2] : Mounting from axle to wheels;
}
```

If a binary Connector declaration includes only the `relatedFeatures` part, then the keyword **`from`** can be omitted.

```
class WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;
    connector axle to wheels;
}
```

If a binary Connector has a single `type`, then the names of the `associationEnds` of the `type` can also be used in the declaration of the `connectorEnds` in the special notation for binary Connectors. However, since the `connectorEnds` are always declared in order from `source` to `target` in this notation, the `associationEnd` names given must match those from the type in the order they are declared for that `type`.

```
class WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;
    connector mount[2] : Mounting
        from mountingAxle => axle
          to mountedWheel => wheels;
}
```

In any of the above notations, a `multiplicity` can be specified for a `connectorEnd`, after the qualified name of the `relatedFeature` for that end. In this case, the given `multiplicity` redefines the `multiplicity` that would otherwise be inherited from the `associationEnd` corresponding to the `connectorEnd`.

```
class WheelAssembly {
    composite feature halfAxles[2] : Axle;
    composite feature wheels[2] : Wheel;
```

```
        // Connects each one of the halfAxles to a different one of the wheels.
        connector mount : Mounting from halfAxles[1] to wheels[1];
    }
```

#### 7.4.4.2.2 Binding Connectors

```
BindingConnector (m : Membership) : BindingConnector =
    ( isAbstract ?= 'abstract' )? 'binding'
    BindingConnectorDeclaration(this, m) TypeBody(m)

BindingConnectorDeclaration (c : BindingConnector, m : Membership) =
    ( FeatureDeclaration(c, m)? 'of' | c.isSufficient ?= 'all' 'of'? )?
    c.ownedFeatureMembership += ConnectorEndMember '='
    c.ownedFeatureMembership += ConnectorEndMember
```

A BindingConnector is declared as a Feature (see 7.3.4.2) using the keyword **`binding`**. In addition, the BindingConnector declaration gives the qualified name of the source `relatedFeature` after the keyword **`of`** and the qualified name of the target relatedFeature after the symbol `=`. If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the Connector is implicitly given a default Subsetting to the Feature *`selfLinks`* from the *Objects* model library (see 8.4). Note that, due to this default subsetting, if no `type` is explicitly given for a BindingConnector, then it will implicitly have the `type` *SelfLink* (the type of *`selfLinks`*).

```
    class Vehicle {
        port feature fuelPort {
            in feature fuelFlow : Fuel;
        }
        composite feature fuelTank {
            in feature fuelIn : fuel;
        }
        binding fuelFlowBinding of fuelPort::fuelFlow = fuelTank::fuelIn;
    }
```

If a BindingConnector declaration includes only the `relatedFeatures` part, then the keyword **`of`** can be omitted.

```
    class Vehicle {
        port feature fuelPort {
            in feature fuelFlow : Fuel;
        }
        composite feature fuelTank {
            in feature fuelIn : fuel;
        }
        binding fuelPort::fuelFlow = fuelTank::fuelIn;
    }
```

As for `connectorEnd` on regular Connectors, redefining multiplicities can also be defined for the `connectorEnds` of BindingConnectors.

### 7.4.4.2.3 Successions

```
Succession (m : Membership) : Succession =
     ( isAbstract ?= 'abstract' )? 'succession'
     SuccessionDeclaration(this, m) TypeBody(this)

SuccessionDeclaration (s : Succession, m : Membership) : Succession :
     ( FeatureDeclaration(s, m)? 'first' | s.isSufficient ?= 'all' 'first'? )?
     s.ownedFeatureMembership += ConnectorEndMember 'then'
     s.ownedFeatureMembership += ConnectorEndMember
```

A Succession is declared as a Feature (see 7.3.4.2) using the keyword **`succession`**. In addition, the Succession declaration gives the qualified name of the source `relatedFeature` after the keyword **`first`** and the qualified name of the target relatedFeature after the keyword **`then`**. If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the Connector is implicitly given a default Subsetting to the Feature *`successions`* from the *Objects* model library (see 8.4). Note that, due to this default subsetting, if no `type` is explicitly given for a Succession, then it will implicitly have the `type` *HappensBefore* (the type of *`successions`*).

```
behavior TakePicture {
    composite step focus : Focus;
    composite step shoot : Shoot;
    succession controlFlow first focus then shoot;
}
```

If a Succession declaration includes only the `relatedFeatures` part, then the keyword **`first`** can be omitted.

```
behavior TakePicture {
    composite step focus : Focus;
    composite step shoot : Shoot;
    succession focus then shoot;
}
```

As for `connectorEnds` on regular Connectors, constraining `multiplicities` can also be defined for the `connectorEnds` of Successions.

```
behavior TakePicture {
    composite step focus[*] : Focus;
    composite step shoot[1] : Shoot;
    // A focus may be preceded by a previous focus.
    succession focus[0..1] then focus[0..1];
    // A shoot must follow a focus.
    succession focus[1] then shoot[0..1];
    // After a shoot, the behavior is done.
    succession shoot then done;
}
```

### 7.4.4.3 Abstract Syntax

## 7.4.4.3.1 Overview

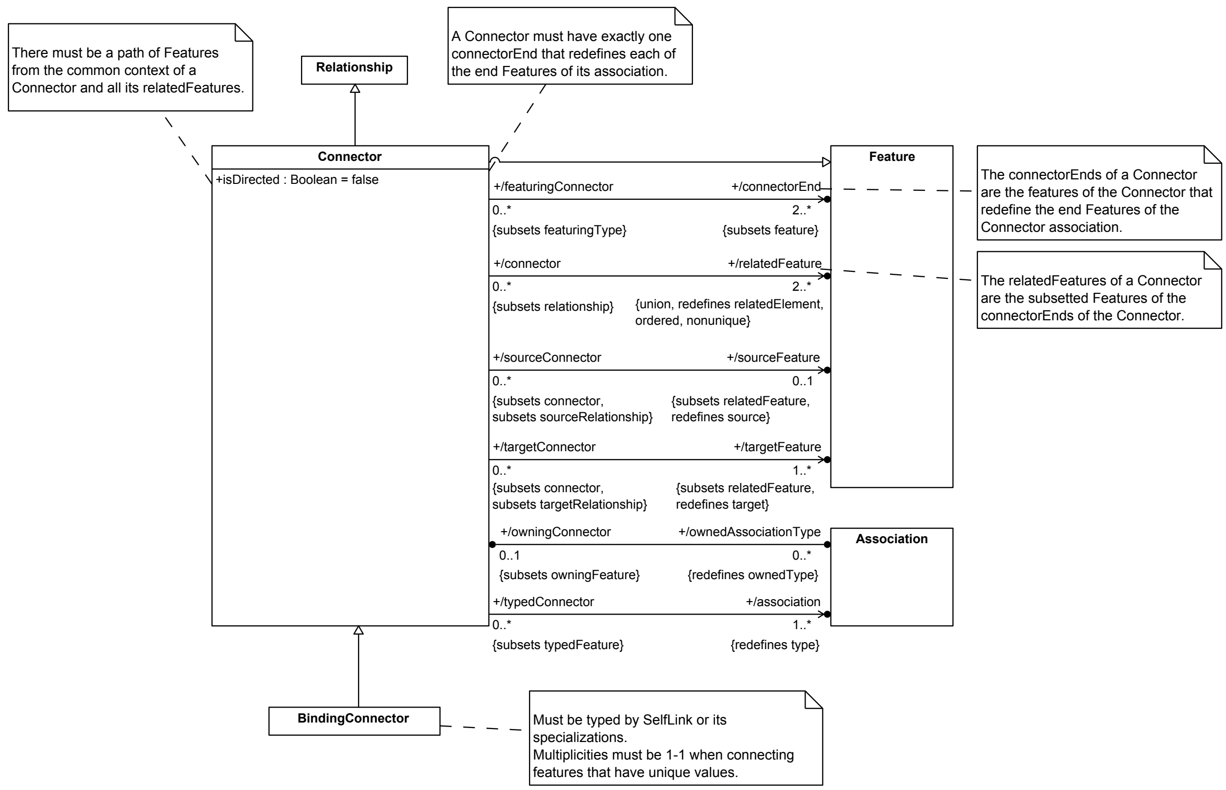


**Figure 19. Connectors**

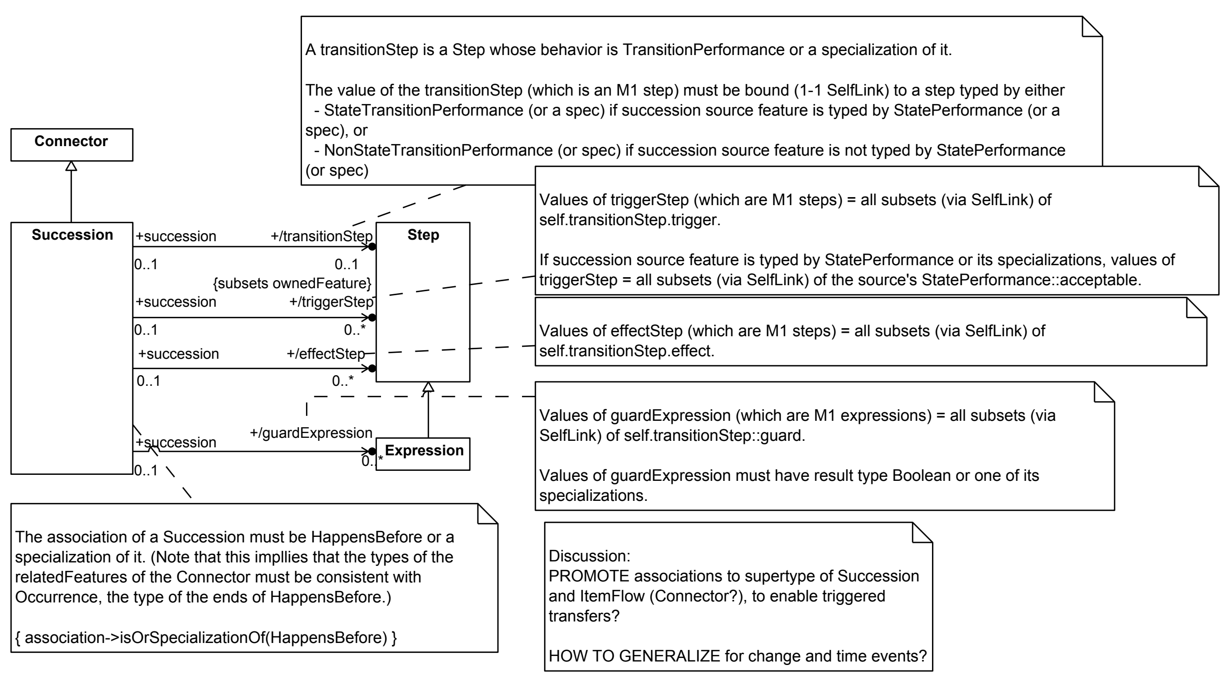


**Figure 20. Successions**

#### 7.4.4.3.2 Binding Connector

**Description**

A Binding Connector is a binary Connector that requires its `relatedFeatures` to identify the same things (have the same values).

**General Classes**

Connector

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.4.3.3 Connector

**Description**

A Connector is a usage of Associations, with links restricted to instances of the Type in which it is used (domain of the Connector). Associations restrict what kinds of things might be linked. The Connector further restricts these links to between values of two Features on instances of its domain.

**General Classes**

Relationship
Feature

**Attributes**

/association : Association [1..*] {redefines type}

The Associations that type the Connector.

/connectorEnd : Feature [2..*] {subsets feature}

The `features` of the Connector that identify the Features that it relates. Each connectorEnd must redefine an `associationEnd` of each of the `associations` of the Connector and subset a single `relatedFeature` of the Connector. A Connector must have at least two `connectorEnds`. A Connector with exactly two `connectorEnds` is known as a *binary* Connector.

isDirected : Boolean

Whether or not the Connector should be considered to have a direction from source to target.

/ownedAssociationType : Association [0..*] {redefines ownedType}

The `associations` of the Connector that are also owned by the Connector.

/relatedFeature : Feature [2..*] {redefines relatedElement, ordered, nonunique, union}

The Features that are related by this Connector considered as a Relationship, derived as the subsetted Features of the `connectorEnds` of the Connector.

/sourceFeature : Feature [0..1] {subsets relatedFeature, redefines source}

The source `relatedFeature` for this Connector. If this is a binary Connector, then the `sourceFeature` is the first `relatedFeature`, and the first `connectorEnd` of the Connector must redefine the `source` Feature of the Connector *`binaryLinks`* from the Kernel Library. If this Connector is not binary, then it has no `sourceFeature`.

/targetFeature : Feature [1..*] {subsets relatedFeature, redefines target}

The target `relatedFeatures` for this Connector. This includes all the `relatedFeatures` other than the `sourceFeature`. If this is a binary Connector, then the `connectorEnds` corresponding to the `targetFeatures` must all redefine the `target` Feature of the Connector *`binaryLinks`* from the Kernel Library.

**Operations**

No operations.

**Constraints**

connectorRelatedFeatures

[no documentation]

```
relatedFeature = connectorEnd.feature
```

### 7.4.4.3.4 Succession

**Description**

A Succession is a binary Connector that requires its `relatedFeatures` to happen separately in time. A Succession must be typed by the Association *HappensBefore* from the Kernel Model Library (or a specialization of it).

**General Classes**

Connector

**Attributes**

/effectStep : Step [0..*]

Steps that represent occurrences that are side effects of the `transitionStep` occurring.

/guardExpression : Expression [0..*]

Expressions that must evaluate to true before the `transitionStep` can occur.

/transitionStep : Step [0..1] {subsets ownedFeature}

A Step that is typed by the Behavior *TransitionPerformance* (from the Model Library) that has this Succession as its `transitionLink`.

/triggerStep : Step [0..*]

Steps that map incoming events to the timing of occurrences of the `transitionStep`. The values of `triggerStep` subset the list of acceptable events to be received by a Behavior or the object that performs it.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.4.4 Semantics

#### Required Generalizations to Model Library

1. Connectors shall (indirectly) specialize *`Objects::links`* (see 8.4.2.4), which means they shall be typed by Associations (7.4.3.3.2).
2. Connector `connectorEnds` shall redefine the `associationEnds` of its `types` and subset its `relatedFeatures`.
3. Connectors `endFeatures` shall all be `connectorEnds`.
4. Connectors with exactly two `relatedFeatures` shall (indirectly) specialize *`Objects::binaryLinks`* (see 8.4.2.2).
5. BindingConnectors shall (indirectly) specialize *`Objects::selfLink`* (see 8.4.2.8), which means they shall be typed by (specializations of) *SelfLink*, or (see 8.4.2.7).
6. Successions shall (indirectly) specialize *`Occurrences::successions`* (see 7.4.4.3.4), which means they shall be typed by (specializations of) *HappensBefore* (see 8.3.2.1) .

#### Connector Semantics

An N-ary Connector of the form

```
connector c : A (f1, f2, ... fN);
```

is semantically equivalent to the Core model

```
feature c : A subsets Objects::links {
    end feature e1 redefines A::e1 subsets f1;
    end feature e2 redefines A::e2 subsets f2;
    ...
    end feature eN redefines A::eN subsets fN;
}
```

where *`e1`*, *`e2`*, ..., *`eN`* are the names of `associationEnds` of the Association *`A`*, in the order they are defined in *`A`*. If explicit `multiplicities` are given for the `connectorEnds`, then these become the `multiplicities` of the `endFeatures` in the semantic model.

If the named notation is used for pairing `connectorEnds` to `associationEnds`:

```
connector c : A (e_f1 => f1, e_f2 => f2, ... e_fN => fN);
```

then the model is similar:

```
feature c : A subsets Objects::links {
    end feature e_f1 redefines A::e_f1 subsets f1;
    end feature e_f2 redefines A::e_f2 subsets f2;
    ...
    end feature e_fN redefines A::e_fn subsets fN;
}
```

where the *e_f1*, *e_f2*, ..., *e_fN* are again names of associationEnds of the Association *A*, but now not necessarily in the order in which they are defined in *A*.

The semantic model of a binary Connector is just that of an N-ary Connector with N = 2. In particular, if no type is explicitly declared for a binary Connector, then its connectorEnds simply redefine the *source* and *target* ends of the Association *BinaryLink*, which are inherited by the Feature *binaryLinks*.

A binary Connector of the form

```
connector from f1 to f2;
```

is semantically equivalent to

```
feature subsets Objects::binaryLinks {
    end feature source redefines Objects::binaryLinks::source subsets f1;
    end feature target redefines Objects::binaryLinks::target subsets f2;
}
```

### Binding Connector Semantics

BindingConnectors are typed by *SelfLinks*, which have two associationEnds that subset each other, meaning they identify the same things (have the same values, see 8.4.2.7). This applies to BindingConnector connectorEnds also by redefining the associationEnds of *SelfLink*.

A BindingConnector of the form

```
binding f1 = f2;
```

is semantically equivalent to the Core model

```
feature subsets Objects::selfLinks {
    end feature self redefines Objects::selfLinks::self subsets f1;
    end feature myself redefines Objects::selfLinks::myself subsets f2;
}
```

where *selfLinks* is typed by *SelfLink* and, so, inherits the endFeatures *self* and *myself*.

### Succession Semantics

Successions are typed by *HappensBefore*, which require the *Occurrence* identified by (value of) its first associationEnd (earlierOccurrence) to precede the one identified by its second (laterOccurrence, see 8.3.2.1). This applies to Succession connectorEnds also by redefining the associationEnds of *HappensBefore*.

A Succession of the form

```
succession first f1 then f2;
```

is semantically equivalent to the Core model

```
feature subsets Occurrences::successions {
    end feature earlierOccurrence
        redefines Occurrences::successions::earlierOccurrence subsets f1;
    end feature laterOccurrenc
        redefines Occurrences::successions::laterOccurrence subsets f2;
}
```

where *`succession`* is typed by *HappensBefore* and, so, inherits the `endFeatures` *`earlierOccurrence`* and *`laterOccurrence`*.

## 7.4.5 Behaviors

### 7.4.5.1 Behaviors Overview

#### Behaviors

Behaviors are Classifiers that classify *Performances* (see 8.5), and how they are related to (other) things in the modeled universe. *Performances* are one kind of *Occurrence*, things that occupy time and space (see 8.3.2.5 and Classes in 7.4.2.1), the other being *Objects* (see 8.4.2.5). Behaviors can coordinate other Behaviors (see Steps below), generate effects on *Objects* involved in them (including their existence and relation to other things), and/or to produce some result before their *Performance* is completed.

Behavior `features` identified as `parameters` are those that specify which *Occurrences* might change their values as the Behavior is carried out:

- *Occurrences* of the Behavior itself (`direction=out`)
- Other *Occurrences* outside of it (`direction=in`)
- Or both (`direction=inout`).

where `direction` is a Feature of a ParameterMembership, a kind of FeatureMembership, that always has a value, defaulting to `in`.

#### Steps

Steps are Features that are typed by Behaviors (their `behaviors`), identifying (having values that are) *Performances* (see 8.5). The `features` of a Behavior that are Steps (the `steps` of the Behavior) specify a refinement of the *Performance* of the Behavior into the *Performances* represented by each of the `steps`. They can be connected by Successions to order their values (which are kinds of *Occurrences*) in time (see 7.4.4). They can also be connected by ItemFlows (see 7.4.8), to model things flowing between `parameters` (`out` or `inout` to in or `inout`).

Steps can inherit the `parameters` of their behaviors or define owned `parameters` to augment or redefine those of their `behaviors`. They can also have nested Steps to augment or redefine the `steps` inherited from their `behaviors`.

### 7.4.5.2 Concrete Syntax

#### 7.4.5.2.1 Behaviors

```
Behavior (m : Membership) : Behavior =
    ( isAbstract ?= 'abstract ')? 'behavior'
    BehaviorDeclaration(this, m) TypeBody(this)

BehaviorDeclaration (b : Behavior, m : Membership) =
    ClassifierDeclaration(b, m) ParameterList(b)?

ParameterList (t : Type) =
    '(' ( t.ownedFeatureMembership += ParameterMember
        ( ',' t.ownedFeatureMembership += ParameterMember )* )? ')'

ParameterMember : ParameterMembership =
    ( direction = FeatureDirection )?
    ownedMemberParameter = ParameterDeclaration(this)

ParameterDeclaration(m : Membership) : Feature =
      FeatureParameterDeclaration(m)
    | StepParameterDeclaration(m)
    | ExpressionParameterDeclaration(m)
    | BooleanExpressionParameterDeclaration(m)

FeatureParameterDeclaration (m : Membership) : Feature =
    'feature'? ( f.isSufficient ?= 'all' )? Identification(this, m)
    ParameterSpecializationPart(this)

StepParameterDeclaration (m : Membership) : Step =
    'step' ( f.isSufficient ?= 'all' )? Identification(this, m)
    ParameterSpecializationPart(this)

ExpressionParameterDeclaration (m : Membership) : Expression =
    'expr' ( f.isSufficient ?= 'all' )? Identification(this, m)
    ParameterSpecializationPart(this)

BooleanExpressionParameterDeclaration (m : Membership) : BooleanExpression =
    'bool' ( f.isSufficient ?= 'all' )? Identification(this, m)
    ParameterSpecializationPart(this)

ParameterSpecializationPart (f : Feature) =
    ParameterSpecialization(f)* MultiplicityPart(f)? ParameterSpecialization(f)*

ParameterSpecialization (f : Feature) =
    TypedBy(f) | Subsets(f) | Redefines(f)
```

A Behavior is declared as a Classifier (see 7.3.3.2.1), using the keyword **`behavior`**. If no `ownedSuperclassing` is explicitly given for the Behavior, then it is implicitly given a default Superclassing to the Behavior *Performance* from the *Performances* library model (see 8.5).

After the Classifier declaration part (including any `ownedSuperclassings`), the Behavior declaration can include a list of owned `parameter` declarations, surrounded by parentheses `(...)`. A `parameter` is declared as a Feature (see 7.3.4.2.1), but the feature keyword is optional. A parameter may also be declared as and Step (see 7.4.5.2.2), Expression (see 7.4.6.2.2) or BooleanExpression (see 7.4.6.2.4) by using the appropriate keyword (**`step`**, **`expr`** or **`bool`**), but without any explicit parameter list for them.

The declaration of a `parameter` can be preceded by a `direction` keyword (**`in`**, **`out`** or **`inout`**). If no `direction` is given explicitly, then the `parameter` has direction **`in`** by default. Other flag keywords (**`abstract`**, **`composite`**, **`portion`**, **`readonly`**, **`derived`**, **`port`**) shall not be used with a `parameter` declaration.

```
// Specializes Objects::Performance by default.
behavior TakePicture (in scene : Scene, out picture : Picture);

behavior RunTest(
    step test : TestProcedure, feature testArtifact : artifact,
    out feature verdict : Verdict);
```

If a Behavior has `ownedSuperclassings` whose `superclasses` are Behaviors, then each of the `ownedParameters` of the subclass Behavior shall, in order, redefine the `parameter` at the same position of each of the `superclass` Behaviors. The redefining `parameters` shall have the same direction as the redefined `parameters`.

```
behavior A ( in a1, out a2);
behavior B ( in b1, out b2);
behavior C specializes A, B
    ( c1 redefines a1 redefines b1, out c2 redefines a2 redefines b2 );
```

If there is a single `superclass` Behavior, then the `subclass` Behavior can declare fewer owned `parameters` than the `superclass` Behavior, inheriting any additional `parameters` from the `superclass` (which are considered to be ordered after any owned `parameters`). If there is more than one `superclass` Behavior, then every `parameter` from every `superclass` must be redefined by an owned `parameter` of the subclass. If every `superclass parameter` is redefined, then the `subclass` Behavior may also declare additional `parameters`, ordered after the redefining `parameters`. If no redefinitions are given explicitly for a `parameter`, then the `parameter` shall be given `ownedRedefinitions` of superclass `parameters` sufficient to meet the previously stated requirements.

```
behavior A1 :> A ( in aa ); // aa redefines A::a1, A::a2 is inherited.
behavior B1 :> B ( in, out, inout b3); // Redefinitions are implicit.
behavior C1 :> A1, B1 (in c1, out c2, inout c3);
```

Steps (see 7.4.5.2.2) declared in the body of a Behavior are the owned `steps` of the containing Behavior. A Behavior can also inherit or redefine non-private `steps` from any `superclass` Behaviors.

```
behavior Focus (in scene : Scene, out image : Image );
behavior Shoot (in image : Image, out picture : Picture);
behavior TakePicture (in scene : Scene, out picture : Picture) {
    composite step focus : Focus (in scene, out image);
    composite step shoot : Shoot (in image, out picture);
}
```

The body of a Behavior is like any other Type body: it contains a list of declarations of `members` of the Behavior treated as a Package. Though the performance of a Behavior takes place over time, the order in which its `steps` are declared has no implication for temporal ordering of the performance of those `steps`. Any restriction on temporal order, or any other connections between the `steps`, must be modeled explicitly.

```
behavior TakePicture (in scene : Scene, out picture : Picture) {
    binding focus::scene = scene;
    composite step focus : Focus (in scene, out image);
    succession focus then shoot;
    composite stream focus::image to shoot::image;
    composite step shoot : Shoot (in image, out picture);
    binding picture = focus::picture;
}
```

#### 7.4.5.2.2 Steps

```
Step (m : Membership) : Step =
    ( isAbstract ?= 'abstract' )? 'step'
    StepDeclaration(this, m) TypeBody(this)

StepDeclaration (s : Step, m : Membership) =
    FeatureDeclaration(s, m) ( ValuePart(s) | StepParameterList(s) )?

StepParameterList (t : Type) =
    '(' ( t.ownedFeatureMembership += StepParameterMember
        ( ',' t.ownedFeatureMembership += StepParameterMember )* )? ')'

StepParameterMember : ParameterMembership =
    ( direction = FeatureDirection )?
    ownedMemberParameter = StepParameter(this)

StepParameter (m : Membership) : Feature =
    ParameterDeclaration(m) ValuePart(this)?
```

A Step is declared as a Feature (see 7.3.4.2) using the keyword **`step`**. If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the Step is implicitly given a default Subsetting to the Feature *`performances`* from the *Performances* library model (see 8.5). Following the Feature declaration part, a Step declaration can include *either* a FeatureValue (see 7.4.9 or a `parameter` list, declared in the same way as for a Behavior (see 7.4.5.2.1).

```
step focus : Focus (in scene, out image);
step shoot : Shoot (in image, out picture);
```

If a Step has `ownedGeneralizations` (including all FeatureTypings, Subsettings and Redefinitions) whose `general` Type is a Behavior or Step, then the rules for the redefinition of the `parameters` of those Behaviors and Steps shall be the same as for the redefinition of the `parameters` of `superclass` Behaviors by a `subclass` Behavior (see 7.4.5.2.1).

```
step focus : Focus
    (in scene, out image); // Parameters redefine parameters of Focus.

step refocus subsets focus; // Parameters are inherited.
```

Unlike the `parameters` declared in a Behavior, the `parameters` of a Step may have FeatureValues (see 7.4.9).

A Step can also have a body, which may have Steps in it. The Step can inherit or override Steps from its Behavior `types` or any other Steps it subsets.

```
step takePictureWithAutoFocus : TakePicture {
    step redefines focus : AutoFocus;
}
```

### 7.4.5.3 Abstract Syntax

#### 7.4.5.3.1 Overview

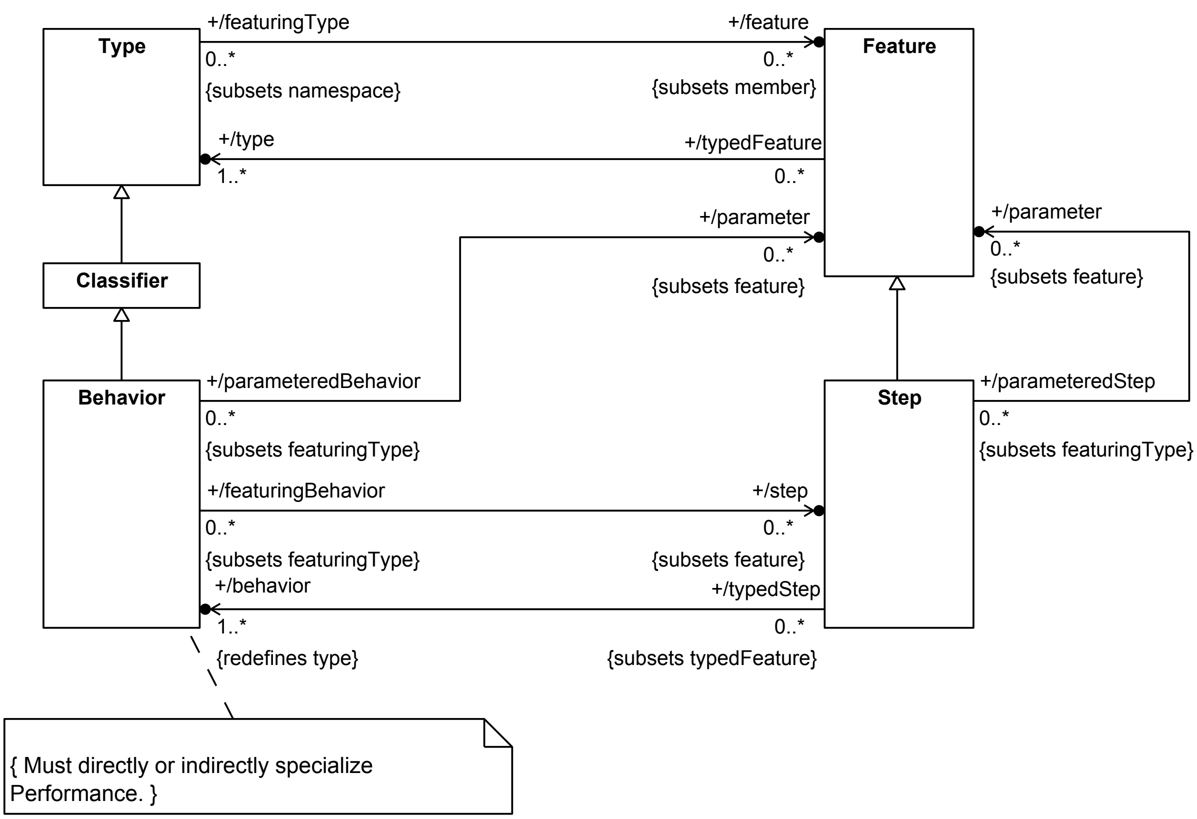


**Figure 21. Behaviors**

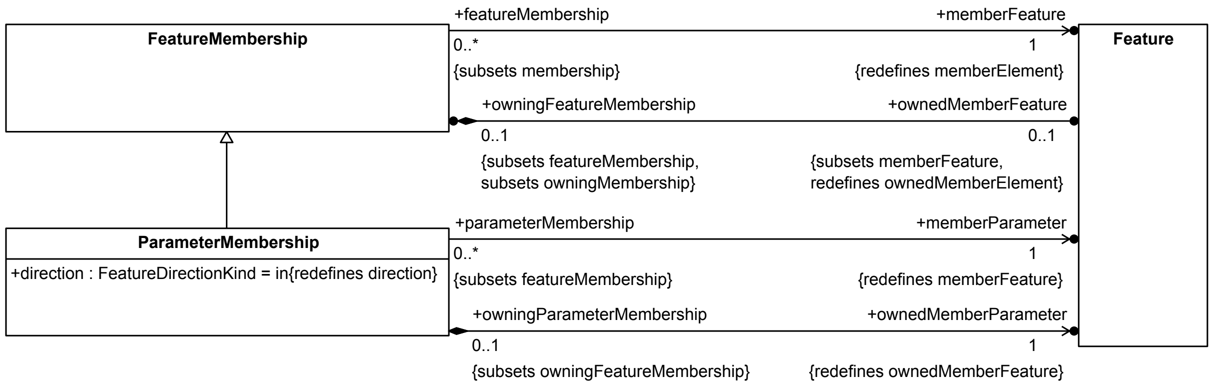


**Figure 22. Parameter Memberships**

### 7.4.5.3.2 Behavior

**Description**

A Behavior coordinates occurrences of other Behaviors, as well as changes in objects. Behaviors can be decomposed into Steps and be characterized by `parameters`.

**General Classes**

Classifier

**Attributes**

/parameter : Feature [0..*] {subsets feature}

The `features` of this Behavior that are owned by the Behavior via ParameterMemberships. A `parameter` always has a `direction`, indicating whether the values of the `parameter` are passed into and/or out of a performance of the Behavior.

/step : Step [0..*] {subsets feature}

The Steps that make up this Behavior.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.5.3.3 Step

**Description**

A Step is a Feature that is typed by one or more Behaviors. Steps may be used by one Behavior to coordinate the performance of other Behaviors, supporting the steady refinement of behavioral descriptions. Steps can be ordered in time and can be connected using ItemFlows to specify things flowing between their parameters.

**General Classes**

Feature

**Attributes**

/behavior : Behavior [1..*] {redefines type}

The Behaviors that type this Step.

/parameter : Feature [0..*] {subsets feature}

The `features` of this Step whose `owningFeatureMemberships` are `ParameterMemberships`. Every `parameter` of a Step must either be inherited from a `behavior` of the Step or directly or indirectly redefine a parameter of a `behavior` of the Step.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.5.3.4 ParameterMembership

**Description**

A ParameterMembership is a FeatureMembership that identifies its `memberFeature` as a parameter, which is always owned. The default `direction` for a ParameterMembership is `in` (unless it is a ReturnParameterMembership). A ParameterMembership must be owned by a Behavior or a Step.

**General Classes**

FeatureMembership

**Attributes**

direction : FeatureDirectionKind {redefines direction}

The default `direction` of a ParameterMembership is `in`.

memberParameter : Feature {redefines memberFeature}

The Feature that is identified as a parameter by this ParameterMembership, which must be the `ownedMemberParameter`.

ownedMemberParameter : Feature {redefines ownedMemberFeature}

The Feature that is identified as a parameter by this ParameterMembership, which is always owned by the ParameterMembership.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.5.4 Semantics

##### Required Generalization to Model Library

1. Behaviors shall (indirectly) specialize *Performances::Performance* (see 8.5.2.10).
2. Steps shall (indirectly) specialize `Performances::performances` (see 8.5.2.11), which means they shall be typed by Behaviors.

##### Behavior Semantics

A Behavior of the form

```
behavior B ( in x, out y, inout z);
```

is semantically equivalent to

```
classifier B specializes Performances::Performance {
    in feature x;
    out feature y;
    inout feature z;
}
```

while a Behavior that explicitly specializes another Behavior:

```
behavior B1 specializes B (in x1, out y1);
```

is semantically equivalent to

```
classifier B1 specializes B {
    in feature x1 redefines x;
    out feature y1 redefines y;
}
```

##### Step Semantics

A Step of the form

```
step s ( in u, out v, inout w);
```

is semantically equivalent to

```
feature s subsets Objects::performances {
    in feature u;
    out feature v;
    inout feature w;
}
```

while a Step that explicitly specializes Behaviors and/or Steps:

```
behavior b : B subsets s (in xx, out yy);
```

is semantically equivalent to

```
feature b : B subsets s {
    in feature xx redefines B::x, s::u;
    out feature yy redefines B::y, s::v;
}
```

**Note.** Steps provide for (repeated) refinement of Behaviors by other Behaviors. The repetition ends with Steps typed by Behaviors from the Kernel Model Library for specifying changes in objects involved in the Behaviors.

**Submission Note.** The Kernel Model Library does not include primitive Behaviors that change objects yet. These will be included in the revised submission.

## 7.4.6 Functions

### 7.4.6.1 Functions Overview

#### Functions

Functions are Behaviors that designate a single output `parameter` as their `result` by a ReturnParameterMembership, a kind of ParameterMembership that requires `direction` = `out`. Functions can have other output `parameters` besides their `result`.

Functions classify *Evaluations* (see 8.5.2.3), which are kinds of *Performances* expected to produce values for their `result`, but can also change involved *Objects* (including creation, destruction, and changes in `feature` values). Some Functions have no parameters with direction `out` or `inout` other than their `result` and do not change objects during their evaluation, such as the numerical functions in the Kernel Model Library (see Clause 8).

#### Expressions

Expressions are Steps typed only by a single Function (their `function`). They can be `steps` in any Behavior, including Functions, in which case such Expression can be designated as specifying the `result` of the Function by a ResultExpressionMembership. The `result` of this Expression shall be connected to the `result` of the featuring Function by a BindingConnector (see 7.4.4).

Expressions can have their own (nested) `parameters`, to augment or redefine those of their `functions`, including the `result`. In particular, they can include another Expression to specify their `result`. In this case, the original Expression must have its own `result`, redefining those from its `function` or any other subsetted Expressions, which must be connected to the `result` of its result Expression by a BindingConnector.

Expressions are commonly organized into tree structures in which the input `parameters` of each Expression are connected by BindingConnectors to the `result` of each of its child Expressions (its `arguments`). KerML textual syntax includes traditional operator notation for constructing such Expression trees (see 7.4.7.2).

#### Predicates

Predicates are Functions that whose `result` is typed by *Boolean* from the *ScalarValues* library (see 8.10) and has a `multiplicity` of (exactly) `1`. Predicates determine whether the values of their input parameters meet particular conditions at the time of evaluation, returning **`true`** if they do, and **`false`** otherwise. They classify *BooleanEvaluations*, which are required to have exactly one `result` of type *Boolean*.

#### Boolean Expressions and Invariants

BooleanExpressions are Expressions whose `function` is a Predicate. As such, a BooleanExpression must similarly have a `result` of type *Boolean.* Invariants are BooleanExpressions all of whose values (which are *BooleanEvaluations*) must `result` in **true**. BooleanExpressions might `result` in **`true`** or **`false`**, but Invariants must always `result` in **`true`**. (BooleanExpressions should not be confused with LiteralBooleans, which are also Expressions with a Boolean `result`, but are not typed by a Predicate and always evaluate to a constant value of **`true`** or **`false`**—see 7.4.7.3.4.)

### 7.4.6.2 Concrete Syntax

#### 7.4.6.2.1 Functions

```
Function (m : Membership) : Function =
    ( isAbstract ?= 'abstract' )? 'function'
    FunctionDeclaration(this, m) FunctionBody(this)

FunctionDeclaration (f : Function, m : Membership) =
    ClassifierDeclaration(f, m) ParameterList(f) ReturnParameterPart(f)

ReturnParameterPart (t : Type) =
    t.ownedFeatureMembership += ReturnParameterMember

ReturnParameterMember : ReturnParameterMembership =
   ownedMemberParameter = ParameterDeclaration(this)

FunctionBody (t : Type) =
    ';' | '{' TypeBodyElement(t)*
              ( t.ownedFeatureMembership += ResultExpressionMember )? '}'

ResultExpressionMember : ResultExpressionMembership =
    ownedResultExpression = OwnedExpression
```

A Function is declared as a Behavior (see 7.4.5.2.1), using the keyword **`function`**, with the addition of the declaration of a `result parameter`. The `result parameter` is declared like any other Behavior `parameter`, but after the parenthesized list of non-result `parameters` for the Function, rather than as part of it. If the Function has no `parameters` other than the result, then an empty set of parentheses `()` shall still be included before the declaration of the `result parameter`. No direction shall be given for a result parameter, since it always has direction out.

```
function Average (scores[1..*] : Rational) : Rational;
function Velocity
    (v_i : VelocityValue, a : AccelerationValue, dt : TimeValue )
    v_f : VelocityValue;
```

If no `ownedSuperclassing` is explicitly given for a Function, then it is implicitly given a default Superclassing to the Function *Evaluation* from the *Performances* library model (see 8.5). If a Function has `ownedSuperclassings` that are Behaviors, then the rules for redefinition or inheritance of non-result parameters shall be the same as for a Behavior (see 7.4.5.2.1). If some of the `superclass` Behaviors are Functions, then the `result parameter` of the subclass Function shall redefine the `result parameters` of the `superclass` Functions. If, in this case, the `result parameter` has no `ownedRedefinitions`, then it shall be implicitly given Redefinitions of the `result parameter` of each of the `superclass` Functions.

```
abstract function Dynamics
    (initialState : DynamicState, time : TimeValue) : DynamicState;
function VehicleDynamics specializes Dynamics
```

```
    // Each parameter redefines the corresponding superclass parameter
    (initialState : VehicleState, time : TimeValue) : VehicleState;
```

The body is like the body of a Behavior (see 7.4.5.2.1), with the optional addition of the declaration of a result Expression at the end. A result Expression shall always be written using the Expression notation described in 7.4.7, *not* using the Expression declaration notation from 7.4.6.2.2.

```
function Average (scores[1..*] : Rational) : Rational {
    import RationalFunctions::Sum;
    import BaseFunctions::Length;

    Sum(scores) / Length(scores)
}
```

**Note.** A result Expression is written *without* a final semicolon.

The `result` of a Function can also be specified using an explicit binding, rather than a result Expression declaration.

```
function Velocity
    (v_i : VelocityValue, a : AccelerationValue, dt : TimeValue )
    v_f : VelocityValue {
    private feature v : VelocityValue = v_i + a * dt;
    binding v_f = v;
}
```

### 7.4.6.2.2 Expressions

```
Expression (m : Membership) : Expression =
    ( isAbstract ?= 'abstract' )? 'expr'
    ExpressionDeclaration(this, m) FunctionBody(this)

ExpressionDeclaration (e : Expression, m : Membership) =
    FeatureDeclaration(e, m)
    ( ValuePart(e) | StepParameterList(e) ReturnParameterPart(e) )?
```

An Expression can be declared as a Step (see 7.4.5.2.2) using the keyword **`expr`** (see also 7.4.7.2 for more traditional Expression notation). If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the Expression is implicitly given a default Subsetting to the Feature *`evaluations`* from the *Performances* library model (see 8.5). Following the Feature declaration part, an Expression declaration can include *either* a FeatureValue (see 7.4.9 or a `parameter` list and `result parameter` part, declared in the same way as for a Function (see 7.4.6.2.1).

```
expr computation : ComputeDynamics (state, dt) result;
expr lastEval : Evaluation = computation;
```

If an Expression has `ownedGeneralizations` (including all FeatureTypings, Subsettings and Redefinitions) whose `general` Type is a Behavior (including a Function) or a Step (including an Expression), then the rules for the redefinition of the `parameters` of those Behaviors and Steps shall be the same as for the redefinition of the `parameters` of `superclass` Behaviors by a `subclass` Function (see 7.4.5.2.1).

```
// Input parameters are inherited, result is redefined.
expr vehicleComputation subsets computation () : VehicleState;
```

As for a generic Step, the `parameters` declared in an Expression declaration may have FeatureValues (see 7.4.9).

An Expression can also have a body which, like a Function body, can specify a result Expression.

```
expr : Dynamics () result : VehicleState {
    vehicleComputation()
}
```

#### 7.4.6.2.3 Predicates

```
Predicate (m : Membership) : Predicate =
    ( isAbstract ?= 'abstract' )? 'predicate'
    PredicateDeclaration(this, m) FunctionBody(this)

PredicateDeclaration (p : Predicate, m : Membership) =
    ClassifierDeclaration(p, m)
    ( ParameterList(p) ReturnParameterPart(p)? )?
```

A Predicate is declared as a Function (see 7.4.6.2.1), using the keyword **`predicate`**, except that declaring the `result parameter` is optional. If a `result parameter` is declared, then it must have `type` *Boolean* from the *ScalarValues* library model (see 8.10) and multiplicity `1..1` (see 7.4.10). If no `result parameter` is declared, then the Predicate is given an implicit one that meets the stated requirements.

```
predicate isAssembled (assembly : Assembly, subassemblies[*] : Assembly);
```

If no `ownedSuperclassing` is explicitly given for a Predicate, then it is implicitly given a default Superclassing to the Predicate *BooleanEvaluation* from the *Performances* library model (see 8.5). If a Predicate has `ownedSuperclassings` that are Behaviors, then the rules for redefinition or inheritance of non-result parameters shall be the same as for a Function (see 7.4.6.2.1).

The body of a Predicate is the same as a Function body (see 7.4.6.2.1). If a result Expression is included, then it shall evaluate to a Boolean `result`.

```
predicate isFull (tank : FuelTank) {
    tank::fuelLevel == tank::maxFuelLevel
}
```

#### 7.4.6.2.4 Boolean Expressions and Invariants

```
BooleanExpression (m : Membership) : BooleanExpression =
    ( isAbstract ?= 'abstract' )? 'bool'
    ExpressionDeclaration(this, m) FunctionBody(this)

Invariant (m : Membership) : Invariant =
    ( isAbstract ?= 'abstract' )? 'inv'
    ExpressionDeclaration(this, m) FunctionBody(this)
```

A BooleanExpression is declared as an Expression (see 7.4.6.2.2), using the keyword **`bool`**, except that declaring the `result parameter` is optional. The requirements on and default for the `result parameter` of a BooleanExpression are the same as for a Predicate (see 7.4.6.2.3). If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the BooleanExpression is implicitly given a default Subsetting to the Feature *`booleanEvaluations`* from the *Performances* library model (see 8.5). If a BooleanExpression has `ownedGeneralizations` (including all FeatureTypings, Subsettings and Redefinitions) whose `general` Type is a Behavior or Step, then the rules for the redefinition of the `parameters` of those Behaviors and Steps shall be the same as for a regular Expression (see 7.4.6.2.2).

```
// All input parameters are inherited.
bool assemblyChecks[*] : isAssembled;
```

A BooleanExpression can also have a body which, like a Predicate body, can specify a *Boolean* result Expression.

```
class FuelTank {
    feature fuelLevel : Real;
    feature readonly maxFuelLevel : Real;
    bool isFull { fuelLevel == maxFuelLevel }
}
```

An Invariant is declared exactly like any other BooleanExpression, except using the keyword **`inv`** instead of **`bool`**.

```
class FuelTank {
    feature fuelLevel : Real;
    feature readonly maxFuelLevel : Real;
    inv { fuelLevel >= 0 & fuelLevel <= maxFuelLevel }
}
```

### 7.4.6.3 Abstract Syntax

#### 7.4.6.3.1 Overview

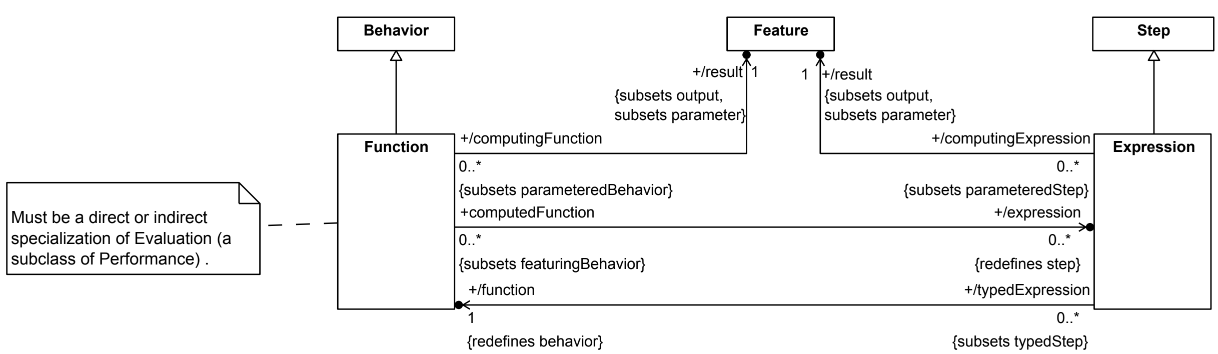


**Figure 23. Functions**

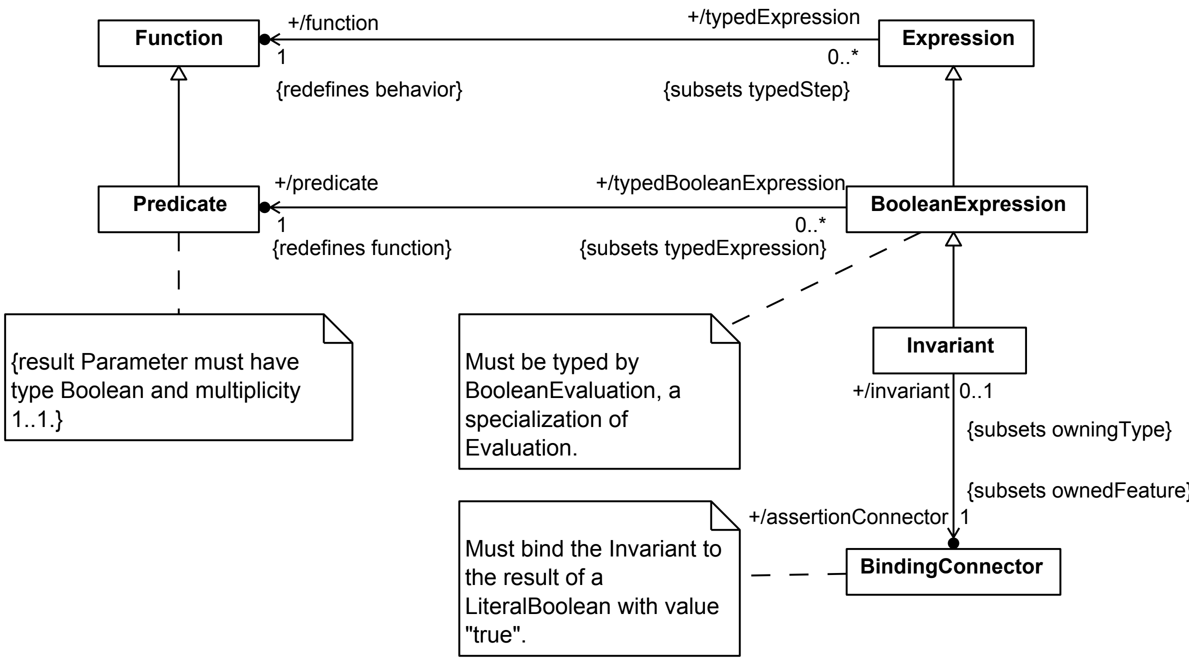


**Figure 24. Predicates**

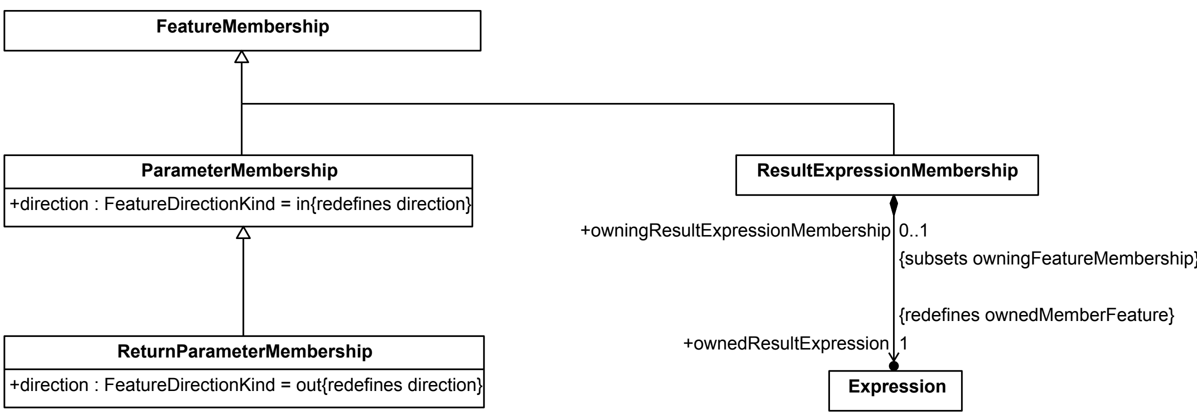


**Figure 25. Function Memberships**

### 7.4.6.3.2 BooleanExpression

**Description**

An BooleanExpression is a Boolean-valued Expression whose type is a Predicate. It represents a logical condition resulting from the evaluation of the Predicate.

A BooleanExpression must subset, directly or indirectly, the Expression *booleanEvaluations* from the Base model library, which is typed by the base Predicate *BooleanEvaluation*. As a result, a BooleanExpression must always be typed by BooleanEvaluation or a subclass of BooleanEvaluation.

**General Classes**

Expression

**Attributes**

/predicate : Predicate {redefines function}

The Predicate that types the Expression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.6.3.3 Expression

**Description**

An Expression is a Step that is typed by a Function. An Expression that also has a Function as its `featuringType` is a computational step within that Function. An Expression always has a single `result` parameter, which redefines the `result` parameter of its defining `function`. This allows Expressions to be interconnected in tree structures, in which inputs to each Expression in the tree are determined as the results of other Expressions in the tree.

**General Classes**

Step

**Attributes**

/function : Function {redefines behavior}

The Function that types this Expression.

/result : Feature {subsets parameter, output}

The single parameter of this Expression whose `owningFeatureMembership` is a ReturnFeatureMembership. The result of an Expression must either be inherited from its `function` or (directly or indirectly) redefine the `result` parameter of its `function`.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.6.3.4 Function

**Description**

A Function is a Behavior that has an output parameter specifically identified as its `result`. It represents the performance of a calculation that produces the values of its `result` parameter. This calculation may be decomposed into Expressions that are the `steps` of the Function.

**General Classes**

Behavior

**Attributes**

/expression : Expression [0..*] {redefines step}

The Expressions that are steps in the calculation of the `result` of this Function.

/result : Feature {subsets parameter, output}

The distinguished `result` parameter of the Function, derived as the `parameter` related to the Function by a ReturnParameterMembership.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.6.3.5 Invariant

**Description**

An Invariant is a BooleanExpression that is asserted to be true. This assertion is made by the Invariant having a BindingConnector as an `ownedFeature` that binds the `result` of the Invariant to the `result` of a LiteralBoolean with value *true*.

**General Classes**

BooleanExpression

**Attributes**

/assertionConnector : BindingConnector {subsets ownedFeature}

An `ownedFeature` of the Invariant that is a BindingConnector between the `result` of the Invariant and the `result` of a LiteralBoolean with value *true*.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.6.3.6 Predicate

**Description**

A Predicate is a Behavior whose `result` Parameter has type *Boolean* and multiplicity 1..1.

**General Classes**

Function

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.6.3.7 ResultExpressionMembership

**Description**

A ResultExpressionMembership is a FeatureMembership that indicates that the `ownedResultExpression` provides the result values for the Function or Expression that owns it. The owning Function or Expression must contain a BindingConnector between the `result` parameter of the `ownedResultExpression` and the `result` parameter of the Function or Expression.

**General Classes**

FeatureMembership

**Attributes**

ownedResultExpression : Expression {redefines ownedMemberFeature}

The Expression that provides the result for the owner of the ResultExpressionMembership.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.6.3.8 ReturnParameterMembership

**Description**

A ReturnParameterMembership is a ParameterMembership that indicates that the `memberParameter` is the `result` parameter of a Function or Expression. The `direction` for a ReturnParameterMembership must be `out`.

**General Classes**

ParameterMembership

**Attributes**

direction : FeatureDirectionKind {redefines direction}

The `direction` of a ReturnParameterMembership must be `out`.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.6.4 Semantics

#### Required Generalizations to Model Library

1. Functions shall (indirectly) specialize *Performances::Evaluation* (see 8.5.2.3).
2. Predicates shall (indirectly) specialize *Performances::BooleanEvaluation* (see 8.5.2.1).
3. Expressions shall (indirectly) specialize *`Performances::evaluations`* (see 8.5.2.4), which means they shall be typed by (specializations of) *Performances::Evaluation*.
4. BooleanExpressions (including Invariants) shall (indirectly) specialize *`Performances::booleanEvaluations`* (see 8.5.2.2), which means they are typed by (specializations of) *Performances::BooleanEvaluation*.

#### Function Semantics

A Function of the form

```
function F (a, b) result {
    resultExpr
}
```

is semantically equivalent to

```
class F specializes Performances::Evaluation {
    in a;
    in b;
    out result redefines Performances::Evaluation::result
        = resultExpr;
}
```

where the binding to *`resultExpr`* is interpreted as a FeatureValue (see 7.4.9).

#### Expression Semantics

An Expression of the form

```
expr e : F (a, b) result {
    resultExpr
}
```

is semantically equivalent to

```
feature e : F subsets Performances::evaluations {
    in a redefines F::a;
```

```
        in b redefines F::b;
        out result redefines F::result
            = resultExpr;
    }
```

### Predicate Semantics

A Predicate is simply a Function with a Boolean result (see 7.4.6.1) and, otherwise, has no additional semantics.

### Boolean Expression and Invariant Semantics

An Invariant of the form

```
    inv i ( ... ) result {
        resultExpr
    }
```

is semantically equivalent to

```
    feature i subsets Performances::booleanEvaluations {
        ...
        out result redefines Performances::booleanEvaluations::result
           = resultExpr;
        private alwaysTrue = true;
        binding result = alwaysTrue;
    }
```

## 7.4.7 Expressions

### 7.4.7.1 Expressions Overview

Expressions are commonly organized into tree structures to specify compound computations (see 7.4.6). KerML includes extensive textual syntax for constructing Expression trees, including traditional operator notations (see 7.4.7.2) for Functions in the Kernel Model Library (see Clause 8). These concrete syntax notations map entirely to an abstract syntax involving just a few specialized Expressions (see 7.4.7.3):

- The non-leaf nodes of an Expression tree are InvocationExpressions, a kind of Expression that specifies its inputs (owned `arguments`) as other Expressions, one for each of the input `parameters` of its `function`.
- The edges of the tree are BindingConnectors between the input `parameters` of an InvocationExpression (redefining those of its `function`) and the `results` of its `argument` Expressions.
- The leaf nodes are these kinds of Expressions:
    - FeatureReferenceExpressions whose `results` are values of a referenced Feature that is not part of the Expression tree, by subsetting the referenced Feature.
    - LiteralExpressions that `result` in the literal value of one of the primitive DataTypes from the *ScalarValues* model library (see 8.10).
    - NullExpressions that `result` in an empty set of values, as required by the `result` multiplicity being 0.

**Submission Note**. Additional operators not currently covered in this Expression syntax include explicit casting, dynamic de-referencing ("dot" expressions) and assignment. These are planned for a revised submission.

#### 7.4.7.2 Concrete Syntax

#### 7.4.7.2.1 Operator Expressions

```
OwnedExpressionMember : FeatureMembership =
    ownedFeatureMember = OwnedExpression

OwnedExpression : Expression =
      ConditionalExpression
    | BinaryOperatorExpression
    | UnaryOperatorExpression
    | ClassificationExpression
    | ExtentExpression
    | SequenceExpression

ConditionalExpression : InvocationExpression =
    ownedFeatureMembership += OwnedExpressionMember
    ownedTyping += [['?']]
    ownedFeatureMembership += OwnedExpressionMember ':'
    ownedFeatureMembership += OwnedExpressionMember

BinaryOperatorExpression : InvocationExpression =
    ownedFeatureMembership += OwnedExpressionMember
    ( ownedTyping += [[BinaryOperator]]
      ownedFeatureMembership += OwnedExpressionMember
    | ownedTyping += [['@']]
      '[' ownedFeatureMembership += OwnedExpressionMember ']'
    )

BinaryOperator =
      '??' | '||' | '&&' | '|' | '^' | '&' | '==' | '!='
    | '<' | '>' | '<=' | '>=' | '+' | '-' | '*' | '/' | '%' | '**'

UnaryOperatorExpression : InvocationExpression =
    ownedTyping += [[UnaryOperator]]
    ownedFeatureMembership += OwnedExpressionMember

UnaryOperator =
    '+' | '-' | '!' | '~'

ClassificationExpression =
    ownedFeatureMembership += OwnedExpressionMember
    ownedTyping += [[ClassificationOperator]]
    ownedFeatureMembership += TypeReferenceMember

ClassificationOperator =
    'istype' | 'hastype'

ExtentExpression : InvocationExpression =
    ownedTyping += ['all']
    ownedFeatureMembership += TypeReferenceMember

TypeReferenceMember : FeatureMembership =
    ownedMemberFeature = TypeReference
```

```
TypeReference : Feature =
    ownedTyping += OwnedFeatureTyping
```

*Operator expressions* provided a shorthand notation for InvocationExpressions that invoke a library Function represented as an *operator symbol*. Table 7 shows the mapping from operator symbols to the Functions they represent from the Kernel Model Library (see Clause 8). An operator expression generally contains subexpressions called its *operands* that generally correspond to the `argument` Expressions of the InvocationExpression, except in the case of operators representing *control Functions*, in which case the evaluation of certain operands is as determined by the Function (see 7.4.7.4 for details).

Operator expressions include the following:

- *Conditional expression.* The *conditional test* operator `?` is a ternary operator that evaluates to the value of its second operand, depending on whether the result of its first operand is true or false. Note that only one of the second or third operand is actually evaluated.

  ```
  x >= 0? x: -x
  ```

- *Binary operator expression.* A *binary operator* is one that has two operands. In general, both operands become arguments of the InvocationExpression, with their results being passed to the invocation of the Function represented by the operator. However, the null-coalescing (`??`), logical and (`&&`) and logical or (`||`) operators all correspond the control Functions (see 8.24) in which their second operand is only evaluated depending on a certain condition of the value of their first operand (whether it is null, true or false, respectively).

  ```
  x + y
  list[i] ?? default
  i > 0 && sensor[i] != null
  ```

- *Unary operator expressions.* A *unary operator* is one that has a single operand. The result of evaluating this operand is passed to the invocation of the Function represented by the operator.

  ```
  -x
  !isOutOfRange(sensor)
  ```

- *Classification expression.* The *classification operators* **`istype`** and **`hastype`** are syntactically similar to binary operators, but, instead of an Expression as their second operand, they take a Type name. These operators test whether the value of their first operand is classified by the named Type (either including or not including subtypes, respectively).

  ```
  sensor istype ThermalSensor
  person hastype Administrator
  ```

- *Extent expression.* The *extent operator* **`all`** is syntactically similar to a unary operator, but, instead of an Expression as its operand, it takes a type name. An extent expression evaluates to a sequence of all instances of the named Type.

  ```
  all Sensor
  ```

Though not directly expressed in the syntactic productions given above, in any operator expression containing nested operator expressions, the nested expressions shall be implicitly grouped according to the *precedence* of the operators involved, as given in Table 8. Operator expressions with higher precedence operators shall be grouped more tightly than those with lower precedence operators. For example, the operator expression

```
-x + y * z
```

is considered equivalent to

```
( (-x) + (y * z) )
```

**Table 7. Operator Mapping**

<table>
<tr><th>Operator</th><th>Library Function</th><th>Description</th></tr>
<tr><td>==</td><td>BaseFunctions::'=='</td><td>Equality</td></tr>
<tr><td>!=</td><td>BaseFunctions::'!='</td><td>Inequality</td></tr>
<tr><td>all</td><td>BaseFunctions::'all'</td><td>Type Extent</td></tr>
<tr><td>istype</td><td>BaseFunctions::'istype'</td><td>Is directly or indirectly instance of type</td></tr>
<tr><td>hastype</td><td>BaseFunctions::'hastype'</td><td>Is directly instance of type</td></tr>
<tr><td>|</td><td>ScalarFunctions::'|'</td><td>Logical "inclusive or"</td></tr>
<tr><td>^</td><td>ScalarFunctions::'^'</td><td>Logical "exclusive or"</td></tr>
<tr><td>&</td><td>ScalarFunctions::'&'</td><td>Logical "and"</td></tr>
<tr><td><</td><td>ScalarFunctions::'<'</td><td>Less than</td></tr>
<tr><td>></td><td>ScalarFunctions::'>'</td><td>Greater than</td></tr>
<tr><td><=</td><td>ScalarFunctions::'<='</td><td>Less than or equal to</td></tr>
<tr><td>>=</td><td>ScalarFunctions::'>='</td><td>Greater than or equal to</td></tr>
<tr><td>+</td><td>ScalarFunctions::'+'</td><td>Addition</td></tr>
<tr><td>-</td><td>ScalarFunctions::'-'</td><td>Subtraction</td></tr>
<tr><td>*</td><td>ScalarFunctions::'*'</td><td>Multiplication</td></tr>
<tr><td>/</td><td>ScalarFunctions::'/'</td><td>Division</td></tr>
<tr><td>%</td><td>ScalarFunctions::'%'</td><td>Remainder</td></tr>
<tr><td>**</td><td>ScalarFunctions::'**'</td><td>Exponentiation</td></tr>
<tr><td>@</td><td>ScalarFunctions::'@'</td><td>Qualification</td></tr>
<tr><td>??</td><td>ControlFunctions::'??'</td><td>Null coalescing</td></tr>
<tr><td>||</td><td>ControlFunctions::'||'</td><td>Conditional "or"</td></tr>
<tr><td>&&</td><td>ControlFunctions::'&&'</td><td>Conditional "and"</td></tr>
<tr><td>?</td><td>ControlFunctions::'?'</td><td>Conditional test (ternary)</td></tr>
</table>

**Table 8. Operator Precedence (highest to lowest)**

| Unary |
|---|
| all |

| |
|---|
| + - ! ~ |
| **Binary** |
| @ |
| ** |
| * / % |
| + - |
| < > <= >= |
| 'istype' 'hastype' |
| == != |
| & |
| ^ |
| \| |
| && |
| \|\| |
| ?? |
| **Ternary** |
| ? |

#### 7.4.7.2.2 Sequence Expressions

```
SequenceExpression : Expression
      SequenceIndexExpression
    | SequenceOperationExpression
    | SequenceConstructionExpression
    | BaseExpression

SequenceIndexExpression : InvocationExpression =
    ownedFeatureMembership += SequenceExpressionMember
    ownedTyping += [['[']]
    ownedFeatureMembership += OwnedExpressionMember ']'

SequenceOperationExpression : InvocationExpression
    ownedFeatureMembership += SequenceExpressionMember
    '->' ownedTyping += [[NAME]]
    ( ownedFeatureMembership += BodyExpressionMember )+

SequenceExpressionMember : FeatureMembership =
    ownedMemberFeature = SequenceExpression

BodyExpressionMember : FeatureMembership =
    ownedMemberFeature = BodyExpression

BodyExpression : Expression =
      ownedFeatureMembership += BodyParameterMember
      ( ownedFeatureMembership += BodyParameterMember )*
      '(' ownedFeatureMembership += OwnedExpressionMember ')'
    | ownedTyping += OwnedFeatureTyping

BodyParameterMember : ParameterMembership =
    memberName = Name ownedMemberFeature = BodyParameter

BodyParameter : Feature =
    ( TypedBy(this) MultiplicityPart(this)? |
      MultiplicityPart(this) TypedBy(this)? )?

SequenceConstructionExpression : Expression =
    '{' SequenceListExpresssion | SequenceRangeExpression '}'

SequenceListExpression : Expression =
    OwnedExpression | SequenceElementList

SequenceElementList : InvocationExpression =
    ownedFeatureMembership += OwnedExpressionMember
    ownedTyping += [[',']]
    ownedFeatureMembership += SequenceListMember

SequenceListMember : FeatureMembership
    ownedFeatureMember = SequenceListExpression

SequenceRangeExpression : InvocationExpression =
    ownedFeatureMembership += OwnedExpressionMember
```

```
ownedTyping += [['..']]
ownedFeatureMembership += OwnedExpressionMember
```

*Sequence expressions* provide a shorthand notation for InvocationExpressions that invoke library Functions that operate on sequences of values. Table 9 shows the mapping from operator symbols and keywords used in sequence expressions to the Functions the represent from the Kernel Model Library (see Clause 8).

Sequence expressions include the following:

- *Sequence index expression.* The first operand of a sequence index expression is expected to evaluate to a sequence of values, and the second operand is expected to evaluate to an index into that sequence. Default indexing is from 1 using *Natural* numbers. However, the functionality of the indexing Function `[` from the *BaseFunctions* library model (see 8.12) may be specialized for domain-specific types.

```
sensors[activeSensorIndex]
```

- *Sequence operation expression.* A sequence operation expression invokes a control Function that takes a sequence as its single argument. The second operand of a sequence operation expression is a parameterized Expression or the name of a Funciton, which can be invoked multiple times as necessary, depending on the functionality of the sequence operation Functions (see 8.24 for further description of these Functions).

```
sensors -> select s (s::isActive)
members -> reject member (!inGoodStanding(member))
factors -> reduce RealFunctions::'*'
```

- *Sequence construction expression.* A sequence construction expression is used to construct a sequence of values, either by giving a list of Expressions that evaluate to the values in the sequence (in order) or by giving two Expressions that evaluate to the upper and lower bounds (inclusive) of a range of values to be included in the sequence.

```
{ fuselage, wing1, wing2, tail }
{ 1 .. size(sensors) }
```

**Table 9. Sequence Operator Mapping**

| Operator | Library Function | Description |
|---|---|---|
| `[` | `BaseFunctions::'['` | Sequence indexing |
| `,` | `BaseFunctions::','` | Sequence construction |
| `collect` | `ControlFunctions::collect` | Sequence collection |
| `select` | `ControlFunctions::select` | Sequence selection |
| `reject` | `ControlFunctions::reject` | Sequence rejection |
| `reduce` | `ControlFunctions::reduce` | Sequence reduction |
| `forAll` | `ControlFunctions::forAll` | Sequence universal test |
| `exists` | `ControlFunctions::exists` | Sequence existential test |
| `..` | `ScalarFunctions::'..'` | Range construction |

#### 7.4.7.2.3 Base Expressions

```
BaseExpression : Expression =
      NullExpression
    | LiteralExpression
    | FeatureReferenceExpression
    | InvocationExpression
    | '(' OwnedExpression ')'

NullExpression : NullExpression =
    'null' | '{' '}'

FeatureReferenceExpression : FeatureReferenceExpression =
    ownedFeatureMembership += FeatureReferenceMember

FeatureReferenceMember : ReturnParameterMembership =
    ownedMemberParameter = FeatureReference

FeatureReference : Feature =
    ownedSubsetting += Subset

InvocationExpression : InvocationExpression =
    ownedTyping += OwnedFeatureTyping '(' ArgumentList(this)? ')'

ArgumentList (e : InvocationExpression) =
    PositionalArgumentList(e) | NamedArgumentList(e)

PositionalArgumentList (e : InvocationExpression) =
    e.ownedFeatureMembership += OwnedExpressionMember
    ( ',' e.ownedFeatureMembership += OwnedExpressionMember )*

NamedArgumentList (e : InvocationExpression) =
    e.ownedFeatureMembership += NamedExpressionMember
    ( ',' e.ownedFeatureMembership += NamedExpressionMember )*

NamedExpressionMember : FeatureMembership =
    memberName = NAME '=>' ownedMemberFeature = OwnedExpression
```

The *base expressions* include representations for InvocationExpressions, FeatureReferenceExpressions, NullExpressions and LiteralExpressions. Parenthesizing any Expression represented in the concrete syntax of 7.4.7.2 also makes it syntactically a base expression, though it is otherwise equivalent to the contained Expression.

Any InvocationExpression can be directly represented by giving the qualified name for the Function to be invoked followed by a list of `argument` Expressions, surrounded by parentheses `()`. The parentheses must be included, even if the `argument` list is empty.

```
IntegerFunctions::'+'(i, j)
isInGoodStanding(member)
Computation()
```

If the qualified name given for an InvocationExpression resolves to an Expression instead of a Function, then the invocation is taken to be for the `function` of the named Expression, as specialized by that Expression.

```
function UnaryFunction(x : Anything): Anything;
function apply(expr fn : UnaryFunction, value : Anything): Anything {
    fn(value) // Invokes UnaryFunction as specified by parameter fn.
}
```

A FeatureReferenceExpression is represented simply by the qualified name of the Feature being referenced.

```
member
spacecraft::mainAssembly::sensors
sensor::isActive
```

Note that the referenced Feature may be an Expression. The notation for a reference to an Expression is distinguished from the notation for an invocation by not have following parentheses.

```
expr addOne : UnaryFunction(x : Anything): Integer {
    x is type Integer? x + 1: 0
}
feature two = apply(addOne, 1); // "addOne" is a reference to expr addOne
```

A NullExpression is notated by the keyword **null**. A NullExpression always evaluates to a result of "no values", which is equivalent to the empty sequence `{}`.

LiteralExpressions are described in 7.4.7.2.4.

#### 7.4.7.2.4 Literal Expressions

```
LiteralExpression : LiteralExpression =
      LiteralBoolean
    | LiteralString
    | LiteralInteger
    | LiteralReal
    | LiteralUnbounded

LiteralBoolean : LiteralBoolean =
    value = BooleanValue

BooleanValue : Boolean =
    'true' | 'false'

LiteralString : LiteralString
    value = STRING_VALUE

LiteralInteger : LiteralInteger =
    value = DECIMAL_VALUE

LiteralReal : LiteralReal =
    value = RealValue

RealValue : Real =
      DECIMAL_VALUE? '.' ( DECIMAL_VALUE | EXPONENTIAL_VALUE )
    | EXPONENTIAL_VALUE

LiteralUnbounded : LiteralUnbounded =
    '*'
```

A LiteralExpression is represented by giving a lexical literal for the `value` of the LiteralExpression.

- A LiteralBoolean is represented by either of the keyword **`true`** or **`false`**.
- A LiteralString is represented by a lexical string value as specified in 7.1.2.6.
- A LiteralInteger is represented by a lexical decimal value as specified in 7.1.2.5. Note that notation is only provided for non-negative integers (i.e., natural numbers). Negative integers can be represented by applying the unary negation operator **-** (see 7.4.7.2.1) to an unsigned decimal literal.
- A LiteralReal is represented with a syntax constructed from lexical decimal values and exponential values (see 7.1.2.5). The full real number notation allows for a literal with a decimal point, with or without an exponential part, as well as an exponential value without a decimal point.

```
3.14
.5
2.5E-10
1E+3
```

- A LiteralUnbounded is represented by the symbol `*`.

### 7.4.7.3 Abstract Syntax

### 7.4.7.3.1 Overview

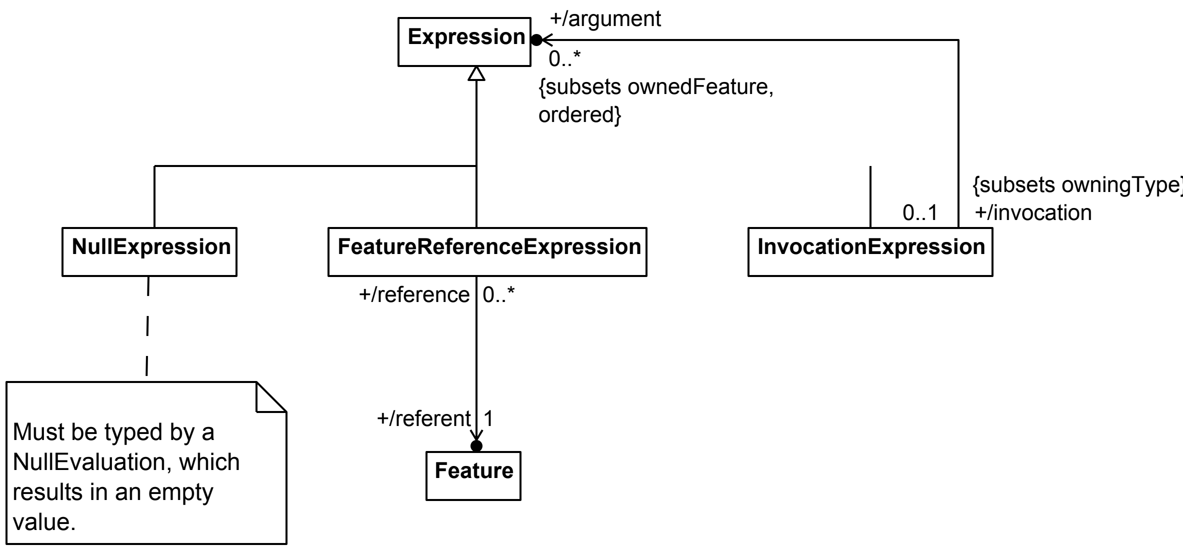


**Figure 26. Expressions**

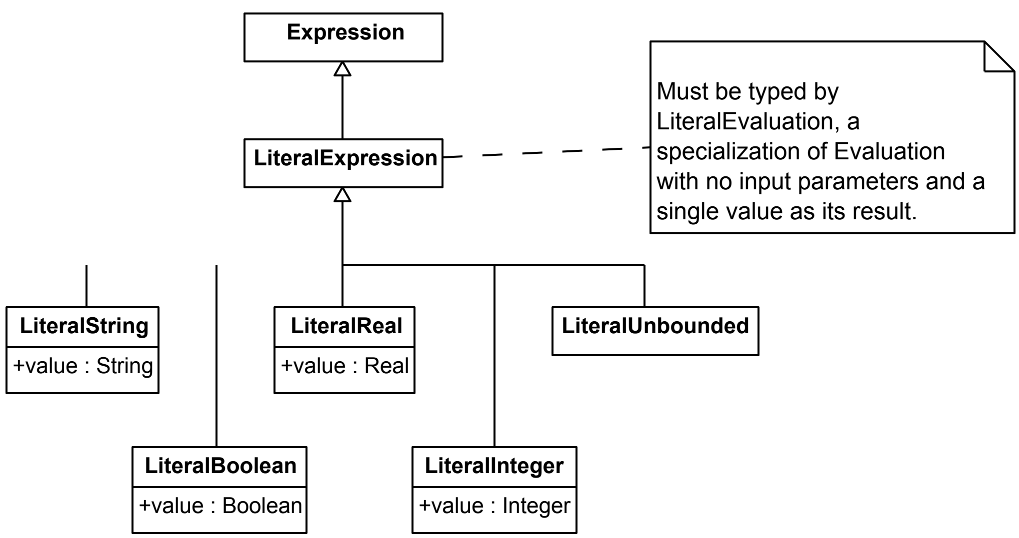


**Figure 27. Literal Expressions**

### 7.4.7.3.2 FeatureReferenceExpression

**Description**

A FeatureReferenceExpression is an Expression whose `result` is bound a `referent` Feature. The only `members` allowed for a FeatureReferenceExpression are the `referent`, the `result` and the BindingConnector between them.

**General Classes**

Expression

**Attributes**

/referent : Feature

The Feature that is referenced by this FeatureReferenceExpression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.3.3 InvocationExpression

**Description**

An InvocationExpression is an Expression each of whose input `parameters` are bound to the `result` of an owned `argument` Expression. Each input `parameter` may be bound to the `result` of at most one `argument`.

**General Classes**

Expression

**Attributes**

/argument : Expression [0..*] {subsets ownedFeature, ordered}

An Expression owned by the InvocationExpression whose `result` is bound to an input `parameter` of the InvocationExpression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.3.4 LiteralBoolean

**Description**

An Expression that provides a *Boolean* value as a result. A LiteralBoolean must have an owned `result` parameter whose type is *Boolean*.

**General Classes**

LiteralExpression

**Attributes**

value : Boolean

The Boolean value that is the result of evaluating this Expression.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.7.3.5 LiteralExpression

**Description**

An Expression that provides a basic value as a result. A LiteralExpression must directly or indirectly specialize the Function *LiteralEvaluation* from the *Base* model library, which has no parameters other than its result, which is a single *DataValue*.

**General Classes**

Expression

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.7.3.6 LiteralInteger

**Description**

An Expression that provides an Integer value as a result. A LiteralInteger must have an owned `result` parameter whose type is *Integer*.

**General Classes**

LiteralExpression

**Attributes**

value : Integer

The Integer value that is the result of evaluating this Expression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.3.7 LiteralReal

**Description**

An Expression that provides a Real value as a result. A LiteralInteger must have an owned `result` parameter whose type is *Real*.

**General Classes**

LiteralExpression

**Attributes**

value : Real

The Real value that is the result of evaluating this Expression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.3.8 LiteralString

**Description**

An Expression that provides a String value as a result. A LiteralString must have an owned `result` parameter whose type is *String*.

**General Classes**

LiteralExpression

**Attributes**

value : String

The String value that is the result of evaluating this Expression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.3.9 LiteralUnbounded

**Description**

An Expression that provides the unbounded ("*") value of the DataType *UnlimitedNatural*. A LiteralUnbounded must have an owned `result` parameter whose type is *UnlimitedNatural*.

**General Classes**

LiteralExpression

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.3.10 NullExpression

**Description**

An Expression that results in a null value. A NullExpression must be typed by a *NullEvaluation* that results in an empty value.

**General Classes**

Expression

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.7.4 Semantics

#### Required Generalizations to Model Library

1. LiteralExpressions shall (indirectly) specialize *`Performances::literalEvaluations`* (see 8.5), which means their `function` is (a specialization of) *Performances::LiteralEvaluations*.
2. NullExpressions shall (indirectly) specialize *`Performances::nullEvaluations`* (see 8.5), which means `function` is (a specialization of) *Performances::NullEvaluations*.

Also see Required Generalizations for Expressions in 7.4.6.4.

### Invocation Expression Semantics

Given a function of the form

```
function F(a, b, ...) result;
```

an InvocationExpression of the form

```
F(expr_1, expr_2, ...)
```

is semantically equivalent to `e::result`, where the Expression `e` is

```
expr e : F (a, b, ...) result {
    expr e_1 ( ) result {
        ...
    }
    expr e_2 ( ) result {
        ...
    }
    ...
    binding a = e_1::result;
    binding b = e_2::result;
    ...
}
```

and each `e_`*n* is the equivalent of *expr_n* according to this subclause.

With the exception of operators that map to control Functions (see below), the concrete syntax operator Expression notation (see 7.4.7.2.1) is simply special surface syntax for InvocationExpressions of standard library Functions. For example, a unary operator Expression such as

```
 ! expr
```

is equivalent to the InvocationExpression

```
ScalarFunctions::'!' (expr)
```

and a binary operator Expression such as

```
expr_1 + expr_2
```

is equivalent to the InvocationExpression

```
ScalarFunctions::'+' (expr_1, expr_2)
```

where the InvocationExpressions are then semantically interpreted as above.

The + and - operators are the only operators that have both unary and binary usages. However, the corresponding library functions have optional 0..1 multiplicity on their second parameters, so it is acceptable to simply not provide an input for the second argument when mapping the unary usages of these operators.

**Submission Note.** Functions in the library Packages BaseFunctions and ScalarFunctions are extensively specialized in other library Packages to constrain their `parameter types` (e.g., the Package RealFunctions constrains `parameter types` to be *Real*, etc.). The semantics of Function specialization and dynamic dispatch based on parameter types will be addressed in the revised submission.

### Control Function Invocation Semantics

Some Functions in the ControlFunctions library model have owned Expressions. InvocationExpressions apply to these Functions as usual, but they also include owned Expressions that redefine each of the owned Expressions of the Function being invoked. This is reflected in special rules in the synthesis of abstract syntax from concrete syntax for such invocations (see 7.4.7.2).

The ternary condition test operator `?` has one `parameter` and two owned Expressions. Therefore, a conditional test Expression (see 7.4.7.2.1) of the form

```
expr_1 ? expr_2 : expr_3
```

is semantically equivalent to the `result` of

```
expr : ControlFunctions::'?' (test) result {
    composite expr e_1 ( ) result {
        ...
    }
    composite expr e_2 redefines thenValue ( ) result {
        ...    }
    composite expr e_3 redefines elseValue ( ) result {
        ...    }
    bind test = e_1::result;
}
```

The binary conditional logical operators `&&` and `||` have one parameter and one owned Expression. Therefore, a conditional logical Expression (see 7.4.7.2.1) of the form

```
expr_1 && expr_2
```

is semantically equivalent to the `result` of

```
expr : ControlFunctions::'&&' (firstValue) result {
    composite expr e_1 ( ) result {
        ...
    }
    composite expr e_2 redefines secondValue ( ) result {
        ...    }
    bind firstValue = e_1::result;
}
```

Finally, the sequence operations `collect`, `select`, `reject`, `reduce`, `forAll` and `exists` have one parameter and one owned Expression. Therefore, a sequence operation expression (see 7.4.7.2.2) such as

```
expr_1 -> select x ( expr_2 )
```

is semantically equivalent to

```
expr : ControlFunctions::select (collection) result {
    composite expr e_1 ( ) result {
        ...
    }
    composite expr redefines selector (x) result {
        expr_2    }
    bind collection = e_1::result;
}
```

### Feature Reference Expression Semantics

A FeatureReferenceExpression for a Feature `f` is semantically equivalent the Expression

```
expr () result subsets f;
```

**Submission Note.** A tighter semantic interpretation for a FeatureReferenceExpression would be to bind the `result parameter` to the reference Feature. However, if the Feature is identified directly using a qualified name, the common context rule for Connectors (see 7.4.4.1) would disallow  desirable references to certain Features, without those Features being redefined within the context of the FeatureReferenceExpression. This will be better addressed in the revised submission by having dynamic feature path Expressions ("dot" Expressions).

### Literal Expression Semantics

With the exception of LiteralUnbounded, each kind of LiteralExpression has a `value` meta-property of a different primitive Type, which determines the `result` of the Expression. LiteralUnbounded does not have a `value` property, because its `result` is always the "unbounded" value `*` from the standard DataType *`UnlimitedNatural`*.

**Submission Note.** The semantics of literals will be more formally addressed in the revised submission.

### Null Expression Semantics

Invocations of it NullExpressions do not product any `result` values (see rules above and 7.4.7.1).

## 7.4.8 Interactions

### 7.4.8.1 Interactions Overview

### Interactions

Interactions are Behaviors that are also Associations (see 7.4.5 and 7.4.3, respectively), whose instances are *Performances* and also *Links* between *Occurrences* (see 8.5, 8.3, and 8.4). They used to specify how participants affect each other and collaborate.

*Transfers* are a kind of Interaction between two participants, as defined in the Kernel Model Library (see 8.6). They specify when things (`items`) are provided by one *Occurrence* (via one of its Feature) and accepted by another (also via a Feature).

### Item Flows

ItemFlows are Steps that are also binary Connectors (see 7.4.5 and 7.4.4, respectively) typed only by *Transfer* (see 8.6), or a specialization of it. An ItemFlow specifies a transfer of `items` between the *Occurrences* identified by its first `connectorEnd` (`transferSource`) and its second (`transferTarget`).

SuccessionItemFlows are ItemFlows that are also Successions (see 7.4.4), requiring time ordering between their `transferSource`, the transfer *Occurrence*, and their `transferTarget`. That is, the transfer happens in the time between the end of its source *Occurrence* and the start of its target *Occurrence*.

### 7.4.8.2 Concrete Syntax

#### 7.4.8.2.1 Interactions

```
Interaction (m : Membership) : Interaction =
    ( isAbstract ?= 'abstract' )? 'interaction'
    BehaviorDeclaration(this, m) TypeBody(m)
```

An Interaction is declared as a Behavior (see 7.4.5.2.1), using the keyword **`interaction`**. If no `ownedSuperclassing` is explicitly given for the Interaction, then it is implicitly given default Superclassings to *both* the Behavior *Performance* from the *Performances* library model (see 8.5) and the Association *BinaryLink* or the Class *Link* from the *Objects* library model (see 8.4), depending on whether it is a binary Association or not.

As a kind of Behavior, if the Interaction has `ownedSuperclassings` whose `superclasses` are Behaviors, then the rules related to their `parameters` are the same as for any `subclass` Behavior (see 7.4.5.2.1). As a kind of Association, the body of an Interaction must declare at least two `associationEnds`. If the Interaction has `ownedSuperclassings` whose `superclasses` are Associations, the rules related to their `associationEnds` are the same as for any `subclass` Association (see 7.4.3.2).

```
interaction Authorization {
    end feature client[*] : Computer;
    end feature server[*] : Computer;
    composite step login;
    composite step authorize;
    composite succession login then authorize;
}
```

#### 7.4.8.2.2 Item Flows

```
ItemFlow (m : Membership) : ItemFlow =
    ( isAbstract ?= 'abstract' )? 'stream'
    ItemFlowDeclaration(this, m) TypeBody(this)

SuccessionItemFlow (m : Membership) : SuccessionItemFlow =
    ( isAbstract ?= 'abstract' )? 'flow'
    ItemFlowDeclaration(this, m) TypeBody(this)

ItemFlowDeclaration (i : ItemFlow, m : Membership) :
    ( FeatureDeclaration(i, m)
      ( 'of'  i.ownedFeatureMembership += ItemFeatureMember
      | i.ownedFeatureMembership += EmptyItemFeatureMember )
      'from'
    | ( isSufficient ?= 'all' )?
      i.ownedFeatureMembership += EmptyItemFeatureMember
    )
    i.ownedFeatureMembership += ItemFlowEndMember 'to'
    i.ownedFeatureMembership += ItemFlowEndMember

ItemFlowFeatureMember : FeatureMembership =
    ( memberName = NAME ':' )? ownedMemberFeature = ItemFeature

ItemFeature : Feature =
      ownedTyping += OwnedFeatureTyping
      ( ownedFeatureMembership += MultiplicityMember )?
    | ownedFeatureMembership += MultiplicityMember
      ( ownedTyping += OwnedFeatureTyping )?

EmptyItemFeatureMember : FeatureMembership =
    ownedMemberFeature = EmptyItemFeature

EmptyItemFeature : Feature =
    {}

ItemFlowEndMember : FeatureMembership =
    ownedMemberFeature = ItemFlowEnd

ItemFlowEnd : Feature =
    ownedFeatureMembership += ItemFlowFeatureMember

ItemFlowFeatureMember : FeatureMembership =
    ownedMemberFeature = ItemFlowFeature

ItemFlowFeature : Feature =
    ownedRedefinition += Redefinition
```

An ItemFlow declaration is syntactically similar to a binary Connector declaration (see 7.4.4.2.1), using the keyword **`stream`**, or **`flow`** for a SuccessionItemFlow. However, rather than specifying the `relatedFeatures` for the ItemFlow, the declaration gives the `sourceOutput` Feature for the Transfer after the keyword **`from`** and the targetInput Feature for the Transfer after the keyword **to**. The relatedFeatures are then determined as the owning Features of the Features given in the ItemFlow declaration. It is these relatedFeatures that are constrained to have a common context with the ItemFlow (see 7.4.4 on the common context rule for Connectors), not the Features actually given in the declaration.

```
class Vehicle {
    composite feature fuelTank {
        out feature fuelOut : Fuel;
    }
    composite feature engine {
        in feature fuelIn : Fuel;
    }
    // The ItemFlow actually connects the fuelTank to the engine.
    // The transfer moves Fuel from fuelOut to fuelIn.
    stream fuelFlow from fuelTank::fuelOut to engine::fuelIn;
}
```

An ItemFlow declaration can also include an explicit declaration of the type and/or multiplicity of the items that are flowing, after the keyword **of**. This asserts that any items transferred by the ItemFlow have the declared Type. In the absence of an item declaration, any values may flow across the ItemFlow, consistent with the types of the sourceOutput and targetInput Featuees.

```
stream of flowingFuel : Fuel from fuelTank::fuelOut to engine::fuelIn;
```

If no Feature declaration or item declaration details are included in an ItemFlow declaration, then the keyword **from** may also be omitted.

```
stream fuelTank::fuelOut to engine::fuelIn;
```

**Note.** ItemFlows are also commonly used to move data from the output parameters of one step to the input parameters of another step.

```
behavior TakePicture {
    composite step focus : Focus (out image : Image);
    composite step shoot : Shoot (in image : Image);
    // The use of a SuccessionItemFlow means that focus must complete before
    // the image is transferred, after which shoot can begin.
    flow focus::image to shoot::image;
}
```

If no ownedSubsetting or ownedRedefinition is explicitly given, then the ItemFlow is implicitly given a default Subsetting to the ItemFlow *transfers* from the *Transfers* model library (see 8.6), or to the SuccessItemFlow *flows*, if a SuccessionItemFlow is being declared. If an Expression has ownedGeneralizations (including all FeatureTypings, Subsettings and Redefinitions) whose general Type is a Behavior or a Step, then the rules for the redefinition of the parameters of those Behaviors and Steps shall be the same as for the redefinition of the parameters of superclass Behaviors by a subclass Step (see 7.4.5.2.2).

#### 7.4.8.3 Abstract Syntax

### 7.4.8.3.1 Overview

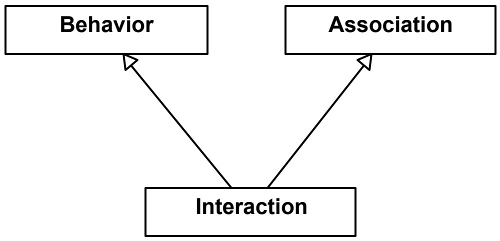


**Figure 28. Interactions**

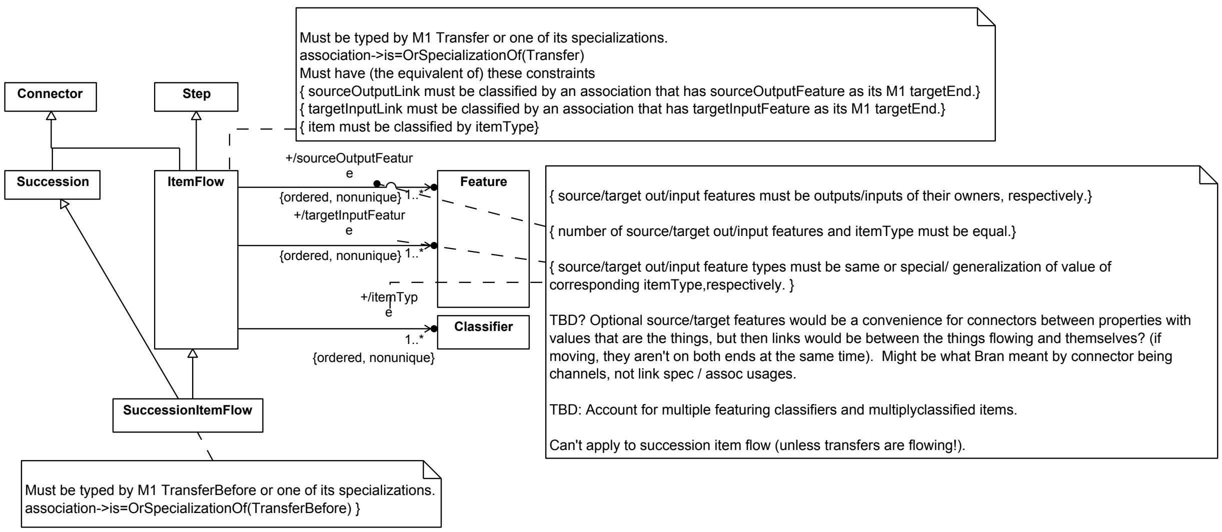


**Figure 29. Item Flows**

### 7.4.8.3.2 ItemFlow

**Description**

An ItemFlow is a Step that represents the transfer of objects or values from one Feature to another. ItemFlows can take non-zero time to complete.

**General Classes**

Connector
Step

**Attributes**

/itemFeature : ItemFeature [1..*] {subsets ownedFeature}

The Feature representing the Item in transit between the source and the target during the transfer. (IMPL)

/itemFlowEnd : ItemFlowEnd [2..*] {redefines connectorEnd}

A `connectorEnd` of this ItemFlow. (IMPL)

/itemFlowFeature : ItemFlowFeature [2..*]

The `sourceOutputFeatures` and `targetInputFeatures` of this ItemFlow. (IMPL).

/itemType : Classifier [1..*] {ordered, nonunique}

The Type of the item transferred.

/sourceOutputFeature : Feature [1..*] {ordered, nonunique}

The Feature that originates the ItemFlow.

/targetInputFeature : Feature [1..*] {ordered, nonunique}

The Feature that receives the ItemFlow.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.8.3.3 Interaction

**Description**

An Interaction is a Behavior that is also an Association, providing a context for multiple objects that have behaviors that impact one another.

**General Classes**

Behavior
Association

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

#### 7.4.8.3.4 SuccessionItemFlow

**Description**

A SuccessionItemFlow is an ItemFlow that also provides temporal ordering. It classifies *Transfers* that cannot start until the source *Occurrence* has completed and that must complete before the target *Occurrence* can start.

**General Classes**

Succession
ItemFlow

**Attributes**

No attributes.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.8.4 Semantics

#### Required Generalizations to Model Library

1. Interactions (indirectly) specialize *Objects::Link* (see 8.4.2.3), or *Objects::BinaryLink* (see 8.4.2.1) for Interactions with exactly two participants.
2. Interactions shall (indirectly) specialize *Performances::Performance* (see 8.5.2.10).
3. ItemFlows shall (indirectly) specialize `Transfers::transfers` (see 8.6.2.2), which means they shall be typed by (specializations of) *Transfers::Transfer* (see 8.6.2.1).
4. The `connectorEnds` of ItemFlows shall
    a. redefine `transferSource` and `transferTarget` of Transfers::Transfer (see 8.6.2.1).
    b. nest Features that redefine `transferSource::sourceOutput` and `transferTarget::targetInput`, respectively.
    c. subset the Features given in the ItemFlow.
5. ItemFlows that specify the kind of item flowing shall add an `ownedFeature` that redefines (specializations of) `Transfer::item`.
6. SuccessionItemFlows (indirectly) specialize `Transfers::flows` (see 8.6.2.4), which means they shall be typed by *Transfers::TransferBefore* (see 8.6.2.3).

#### Interaction Semantics

An Interaction of the form

```
interaction I (in x, out y, inout z) {
    end feature e1;
    end feature e2;
}
```

is semantically equivalent to the Core model

```
class I specializes Objects::BinaryLink, Performances::Performance {
    end feature e1 redefines Objects::BinaryLink::source {
        feature e2 = I::e2(e1);
    }
    end feature e2 redefines Objects::BinaryLink::target {
        feature e1 = I::e2(e2);
    }
    in feature x;
    out feature y;
```

```
        inout feature z;
    }
```

### Item Flow Semantics

An ItemFlow of the form

```
    stream of item : T from f1::f1_out to f2::f2_in;
```

is semantically equivalent to the core model

```
    feature subsets Transfer::transfers {
        end feature redefines transferSource subsets f1 {
            feature redefines sourceOutput subsets f1::f1_out;
        }
        end feature redefines transferTarget subsets f2 {
            feature redefines targetInput subsets f2::f2_in;
        }
    }
```

## 7.4.9 Feature Values

### 7.4.9.1 Feature Values Overview

FeatureValues are FeatureMemberships that require a Feature to identify (have values that are) the `result` of evaluating a nested Expression (`value` ). Features that have a FeatureValue (at most one) shall also have a nested BindingConnector (`valueConnector`) between the Feature and `result` of the `value` Expression.

### 7.4.9.2 Concrete Syntax

```
ValuePart (f : Feature) =
    '=' f.ownedFeatureMembership += FeatureValue

FeatureValue : FeatureValue =
    value = OwnedExpression
```

A Feature value is is declared using the symbol = followed by a representation of the value Expression using the concrete syntax from 7.4.7.2. This notation is appended to the declaration of the Feature that is the `featureWithValue` for the FeatureValue. A FeatureValue can be included with the following kinds of Feature declaration:

- Feature (see 7.3.4.2.1)
- Step (see 7.4.5.2.2)
- Expression (see 7.4.6.2.2)
- BooleanExpression and Invariant (see 7.4.6.2.4)

```
    feature monthsInYear : Natural = 12;
    class TestRecord {
        feature scores[1..*] : Integer;
        derived feature averageScore[1] : Integer = sum(scores)/size(scores);
    }
```

**Note.** Despite the similarity of the notation to use in some previous modeling languages, a FeatureValue in KerML is *not* a "default value" or an "initial value". Instead, the semantics of binding mean that a FeatureValue asserts that a Feature is *equivalent* to the result of the value Expression (see 7.4.4.4 on the semantics of BindingConnectors). To highlight this, a feature of a Type with a FeatureValue can be flagged as **`derived`** (though this is not required, nor is it required that the value of a **`derived`** Feature be computed using a FeatureValue).

**Submission Note.** Allowing the specification of default and initial values is planned to be addressed in the revised submission.

A FeatureValue can also be used in the declaration of a `parameter` of in a Step or Expression declaration (see 7.4.5.2.2 and 7.4.6.2.2).

```
behavior ProvidePower(in cmd : Command, out wheelTorque : Torque) {
    composite step generate : GenerateTorque(
        in cmd = ProvidePower::cmd,
        out generatedTorque);
    composite step apply : ApplyTorque(
        in generatedTorque = generate::generatedTorque,
        out appliedTorque = ProvidePower::wheelTorque);
}
```

### 7.4.9.3 Abstract Syntax

#### 7.4.9.3.1 Overview

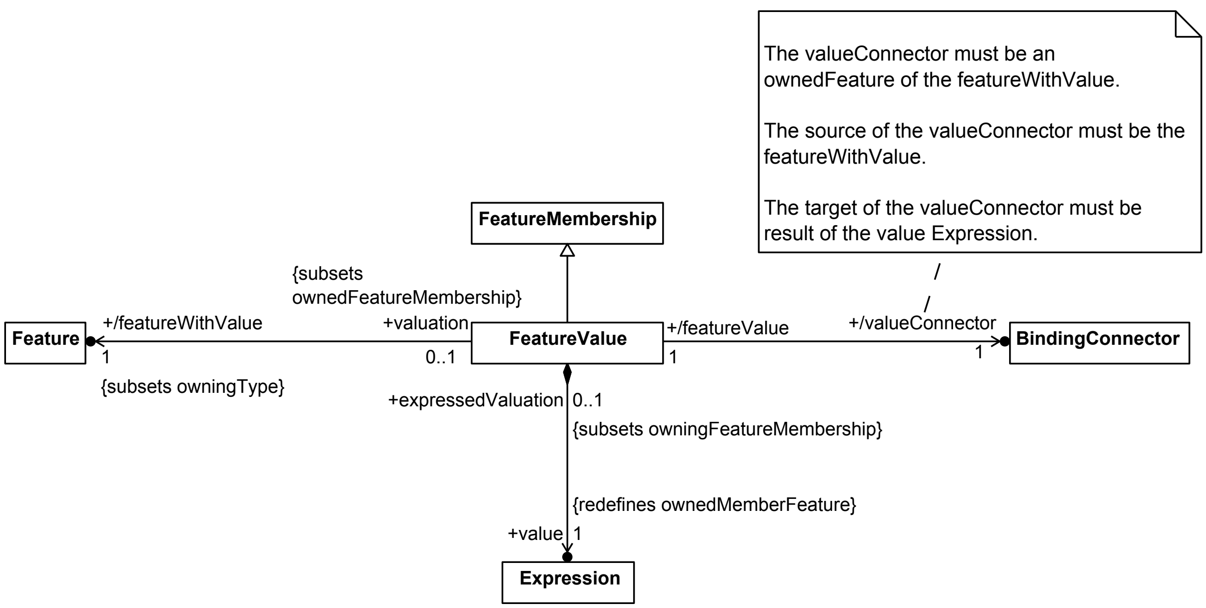


**Figure 30. Feature Values**

#### 7.4.9.3.2 FeatureValue

**Description**

A FeatureValue is a FeatureMembership that identifies a particular member Expression that provides the value of the Feature that owns the FeatureValue. A FeatureValue requires that there be a BindingConnector between the Feature and the `result` of the Expression, enforcing the intended semantics. A Feature can have at most one FeatureValue.

**General Classes**

FeatureMembership

**Attributes**

/featureWithValue : Feature {subsets owningType}

The Feature to be provided a value.

value : Expression {redefines ownedMemberFeature}

The Expression that provides the value as a result.

/valueConnector : BindingConnector

The BindingConnector that binds the result of the Expression to the Feature. The `valueConnector` must be an `ownedFeature` of the `featureWithValue`. The source of the `valueConnector` must be the `featureWithValue`. The target of the `valueConnector` must be result of the `value` Expression.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.9.4 Semantics

A Feature of the form

```
feature f = expr;
```

is semantically equivalent to

```
feature f {
    expr e () result { ... }
    binding f = e::result;
}
```

where `e` is the interpretation of *`expr`* as described in 7.4.7.4.

## 7.4.10 Multiplicities

### 7.4.10.1 Multiplicities Overview

Core defines Multiplicity as a Feature for specifying cardinalities (number of instances) of a Type by enumerating all numbers the cardinality might be (see 7.3.4.3.4). Kernel specializes this to MultiplicityRanges for specifying cardinalities by two natural numbers (a range). A MultiplicityRange has `lowerBound` and `upperBound` Expressions that are evaluated to determine the lowest and highest cardinalities. The `lowerBound` Expression `result` shall be typed by *Natural,* while the `upperBound` Expression shall result `result` shall be typed by *UnlimitedNatural* (both from the ScalarValues library model, see 8.10). An `upperBound` value of * means that the cardinality includes all numbers greater than or equal to the `lowerBound` value (unbounded).

**Submission Note.** More kinds of Multiplicities (such as multiple ranges like `[2..4, 6..8]`) will be considered for the revised submission.

#### 7.4.10.2 Concrete Syntax

```
Multiplicity : MultiplicityRange =
    '[' ( ownedFeatureMembership += OwnedExpressionMember '..' )?
          ownedFeatureMembership += OwnedExpressionMember ']'
```

A MultiplicityRange is written in the form `[`*`lowerBound`*`..` *`upperBound`*`]`, where *`lowerBound`* and *`upperBound`* are Expressions represented in the notation described in 7.4.7. LiteralExpressions can be used to in a MultiplicityRange with fixed lower and/or upper bounds. If only a single Expression is given, then the result of the Expression is used as both the lower and upper bound of the range, unless the result is the UnlimitedNatural unbounded value `*`, in which the lower bound is taken to be 0.

```
class Automobile {
    composite feature wheels : Wheel[4]; // Equivalent to [4..4]
    feature driveWheels[2..4] subsets wheels;
}
feature autoCollection : Automobile[*]; // Equivalent to [0..*]
```

#### 7.4.10.3 Abstract Syntax

#### 7.4.10.3.1 Overview

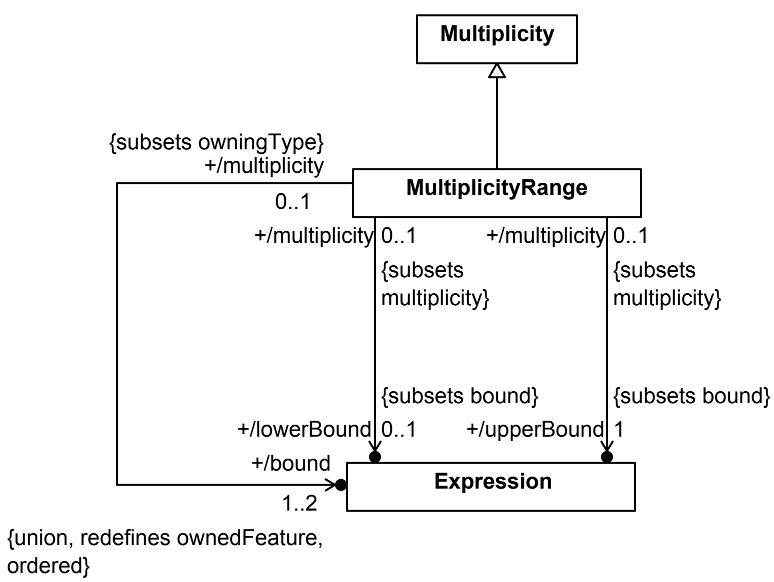


**Figure 31. Multiplicities**

#### 7.4.10.3.2 MultiplicityRange

**Description**

A MultiplicityRange is a Multiplicity whose value is defined to be the (inclusive) range of natural numbers given by the result of a `lowerBound` Expression and the result of an `upperBound` Expression. The result of the `lowerBound` Expression shall be of type *Natural*, while the result of the `upperBound` Expression shall be of type *UnlimitedNatural*. If the result of the `upperBound` Expression is the unbounded value `*`, then the specified range includes all natural numbers greater than or equal to the `lowerBound` value.

**General Classes**

Multiplicity

**Attributes**

/bound : Expression [1..2] {redefines ownedFeature, ordered, union}

The bound Expressions of the MultiplicityRange. These shall be the only `ownedFeatures` of the MultiplicityRange.

/lowerBound : Expression [0..1] {subsets bound}

The Expression whose result provides the lower bound of MultiplicityRange. If no `lowerBound` Expression is given, then the lower bound shall have the same value as the upper bound, unless the upper bound is unbounded (`*`), in which case the lower bound shall be 0.

/upperBound : Expression {subsets bound}

The Expression whose result is the upper bound of the MultiplicityRange.

**Operations**

No operations.

**Constraints**

No constraints.

### 7.4.10.4 Semantics

#### Required Generalizations to Model Library

1. MultiplicityRanges shall directly subset `Base::naturals` (see 8.2), which means they shall be typed by (a specialization of) *ScalarValues::Natural*.

#### Multiplicity Range Semantics

A MultiplicityRange of the form

```
[expr_1.. expr_2]
```

represents a range of data values of the DataType *Natural* (see 8.10.2.5) that are greater than or equal to the result of the Expression `expr_1` and less than or equal to the result of the Expression `expr_2`. Essentially, this is

```
all Natural -> select n (expr_1 <= n & n <= expr_2)
```

where, if `expr_2` evaluates to the unbounded value `*`, all *Natural* data values are less than it.

A MultiplicityRange having only a single expression:

```
[expr]
```

is interpreted in one of the following ways:

- If `expr` evaluates to `*`, then the values of the MultiplicityRange are the entire extent of *Natural.*
- Otherwise, the values of the MultiplicityRange are all *Natural* data values less than or equal to the result of `expr`.
  `all Natural -> select n (n <= expr)`

**Note.** A conforming tool is not expected to compute the entire set of *Natural* numbers that are values of a MultiplicityRange. It is sufficient to check that the values of a Type have a cardinality that is within the range specified by MultiplicityRange.

# 8 Model Library

## 8.1 Model Library Overview

The Kernel Model Library is a collection of KerML models that are part of the semantics of the metamodel (see Clause 7). They are reused when constructing KerML user models (instantiating the metamodel), as specified by constraints and semantics of metaelements, such as Types being required to specialize Anything from the library and Behaviors specializing Performances. The library can be specialized for particular applications, such as systems.

The major areas covered in the Model Library are:

1. The *Base* library model (see 8.2) provides the base of the Generalization hierarchy for all KerML Types, including the most general Classifier *Anything* and the most general Feature `things`. It also contains the base DataType *DataValue* and its corresponding base Feature `dataValues`.
2. The *Occurrences* library model (see 8.3) models *Occurrences* and the temporal Associations between them. The *Objects* library model (see 8.4) then specializes *Occurrences* to provide a model of *Objects* and *Links,* giving semantics to Classes and Associations, respectively. And the *Performances* library model (see 8.5) specializes *Occurrences* to provide a model of *Performances* and *Evaluations*, giving semantics to Behaviors and Expressions, respectively. Temporal associations can therefore be used to specify the order in which Performances are carried out during other Performances, or when Objects exist in relation to each other, or combinations involving Performances and Objects. The *Transfers* library model (see 8.6) models the asynchronous of items between *Occurrences,* giving semantics to Item Flows.
3. The *Control Performances, State Performances* and *TransitionPerformances* library models (see 8.7, 8.8 and 8.9) provide semantic models for the coordination of a number of *Performances* that, together, collaborate to carry out some overall combined *Performance*. KerML does not provide any special syntax corresponding to these library models (e.g., KerML does not have any "control node" or "state machine" syntax), though various Behaviors define in the library models can be used as the `types` of Steps in order to coordinate the *Performance* of an overall containing Behavior. It is expected that other languages built on KerML and using these library models can add syntax as needed by their applications.
4. The *ScalarValues* and *NonScalarValues* model libraries (see 8.10 and 8.11) provide a set of primitive and collection DataTypes that can be used in KerML user models. Additional library models (see 8.12 through 8.24) then provide Functions that can be used to operate both on the standard DataTypes and more generally. The KerML operator and sequence expression notations map to invocations of certain of these library Functions. It is expected that other languages built on KerML will provide additional domain models as needed by their applications, which may include specializations of the library Functions for domain-specific DataTypes. In this way, the same KerML concrete syntax for Expressions notation can continue to be used, extended with domain-specific semantics.

**Note.** Since KerML does not provide any normative graphical notation, the diagrams in this clause should be considered informative overviews of the library models they depict. The normative representation of all library models is using the textual concrete syntax, as provided in machine-readable files associated with this specification document.

**Submission Note.** The documentation provided in this clause is currently incomplete. Full documentation, as appropriate, will be provided in the revised submission.

## 8.2 Base

### 8.2.1 Base Overview

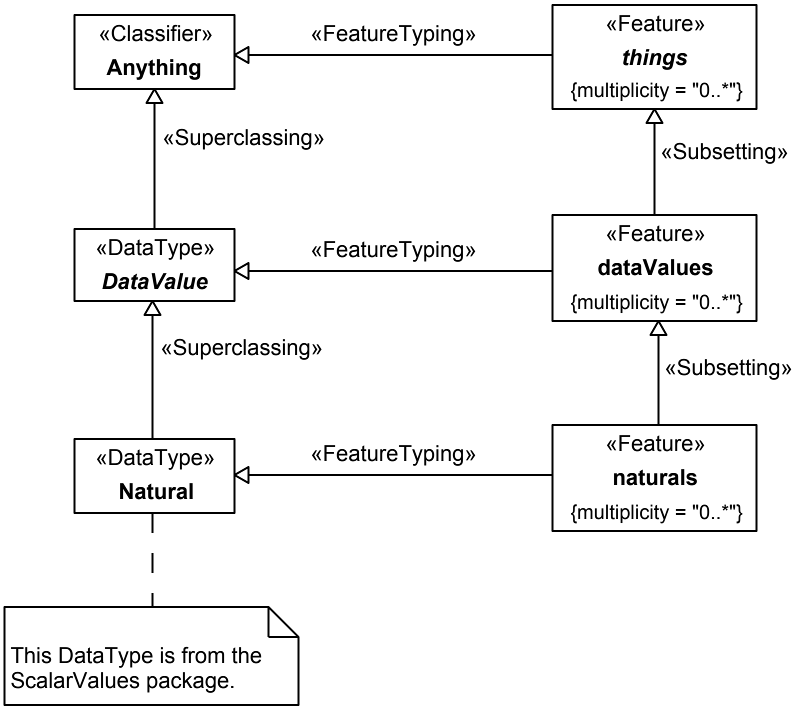


**Figure 32. Base Types**

## 8.2.2 Elements

### 8.2.2.1 Anything <Classifier>

**Description**

Anything is the most general Classifier (M1 instance of M2 Classifier). All other M1 elements (in libraries or user models) specialize it (directly or indirectly). Anything is the FeatureType for `things`, the most general Feature. Since FeatureType is a kind of Generalization, this means that Anything is also a generalization of `things`.

**General Classes**

No general classes.

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.2.2.2 DataValue <DataType>

**Description**

DataValue is Anything that can only be distinguished by how they are related to other things (via Features). It is the most general Datatype (M1 instance of M2 Datatype). All other M1 Datatypes (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

Anything

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.2.2.3 dataValues <Feature>

**Description**

`dataValues` is a specialization of `things` restricted to type `DataValue`. All other Features typed by `DataValue` or its specializations (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

DataValue
things

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.2.2.4 naturals <Feature>

**Description**

**General Classes**

Natural
dataValues

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.2.2.5 things <Feature>

**Description**

`things` is the most general Feature (M1 instance of M2 Feature). All other Features (in libraries or user models) specialize it (subset or redefine, directly or indirectly). It is typed by Anything.

**General Classes**

Anything

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.3 Occurrences

## 8.3.1 Occurrences Overview

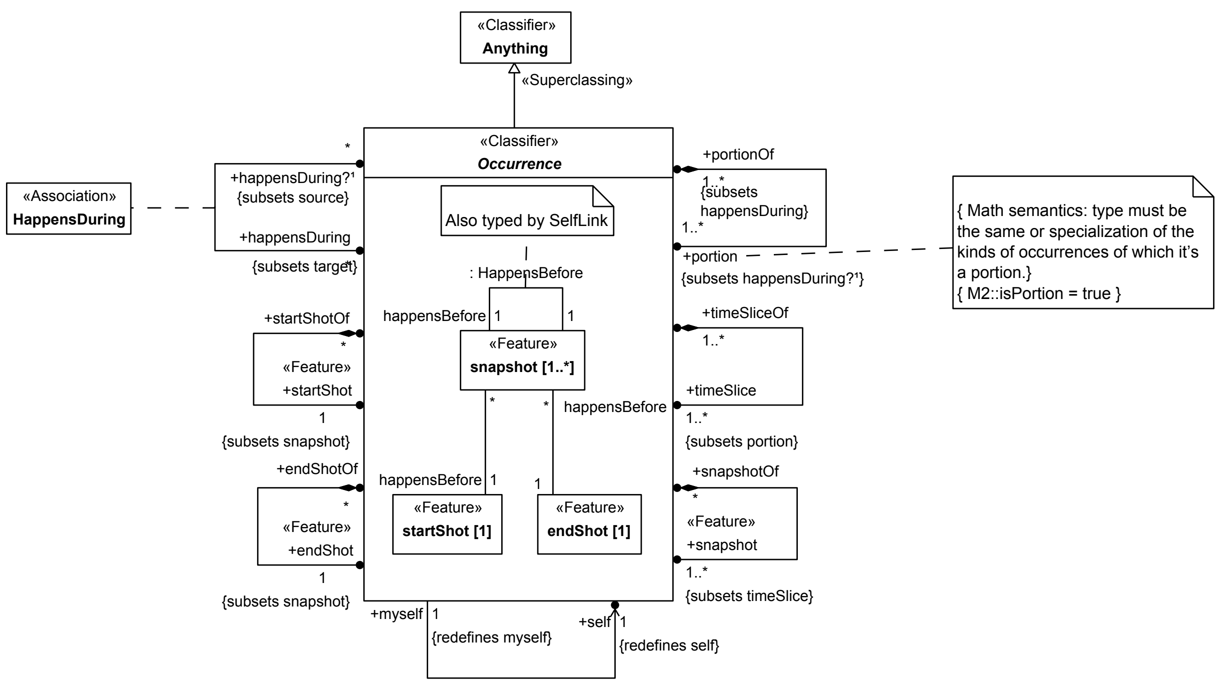


**Figure 33. Occurrences**

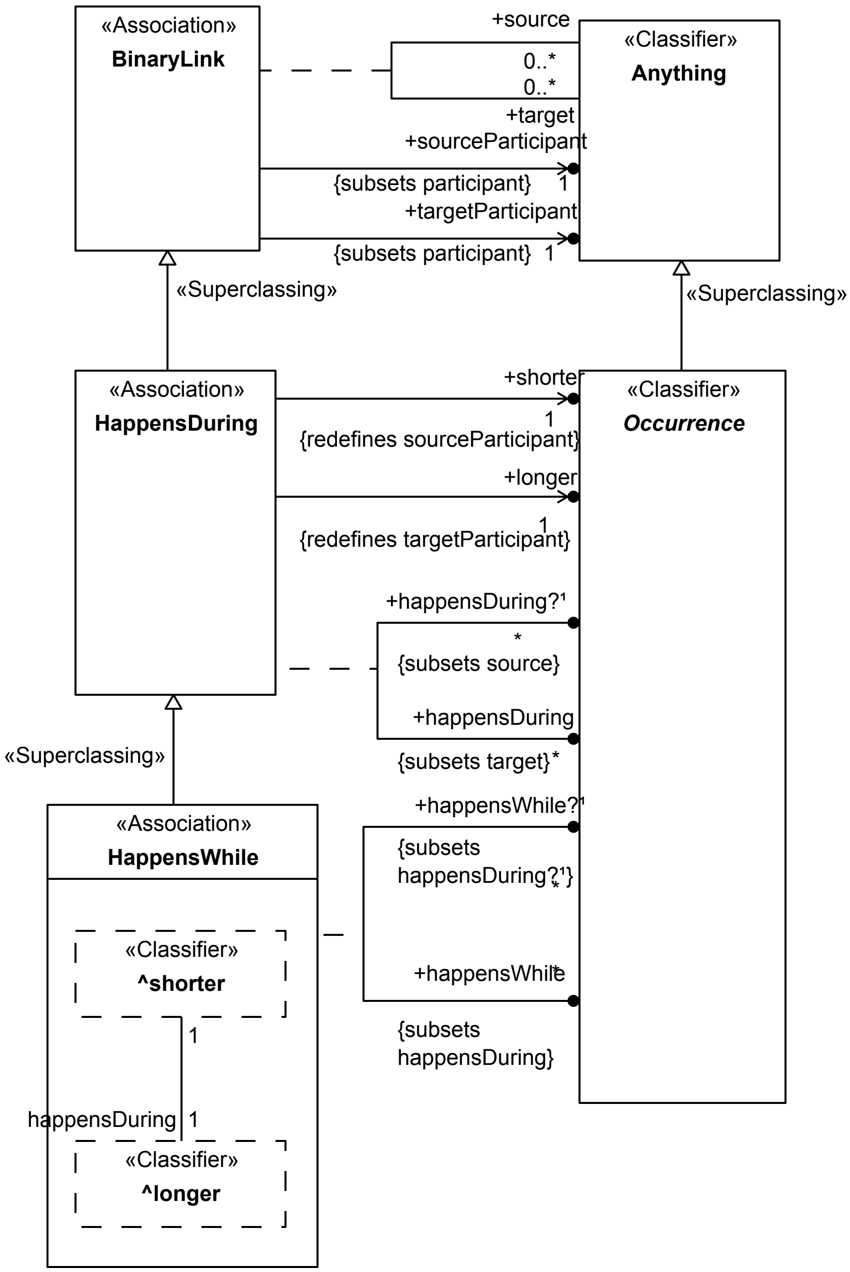


**Figure 34. Happenings During**

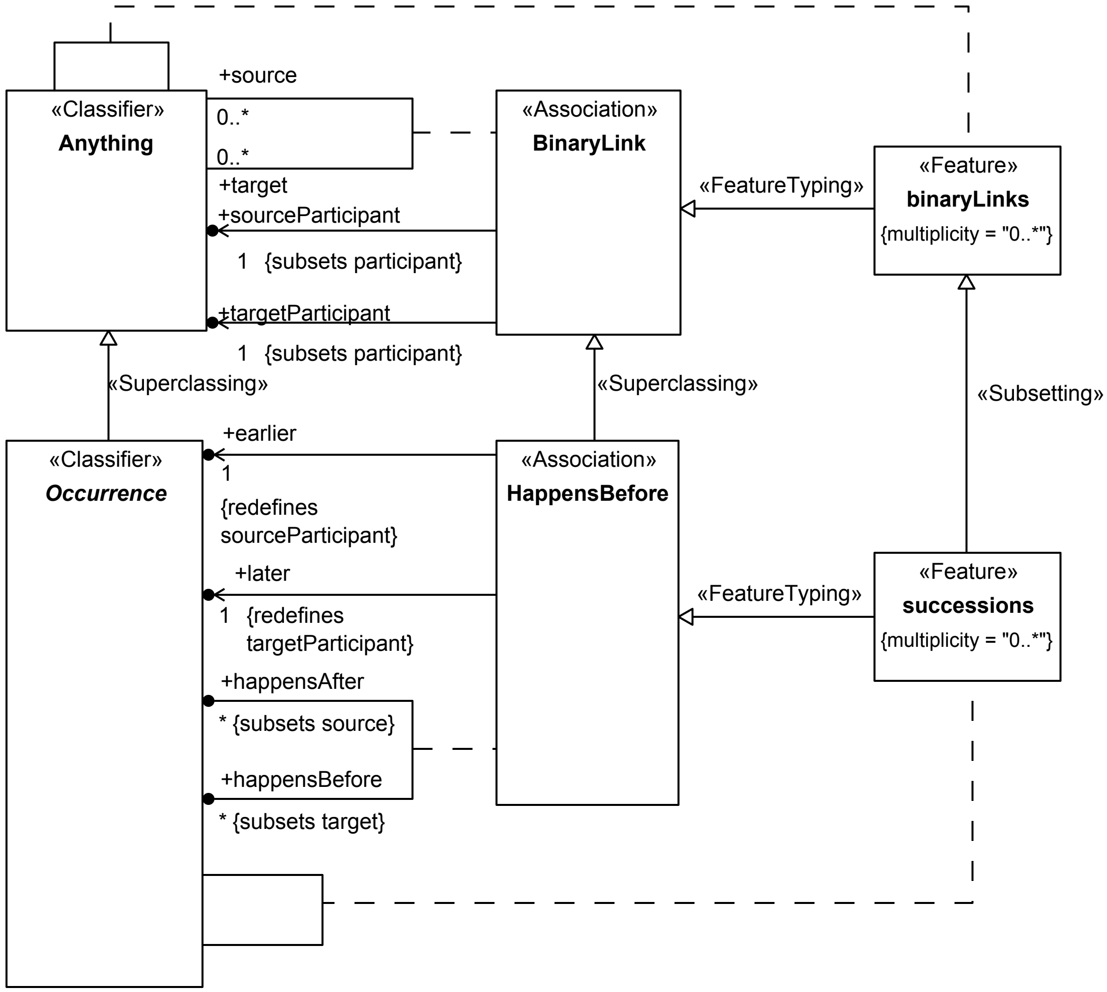


**Figure 35. Happenings Before**

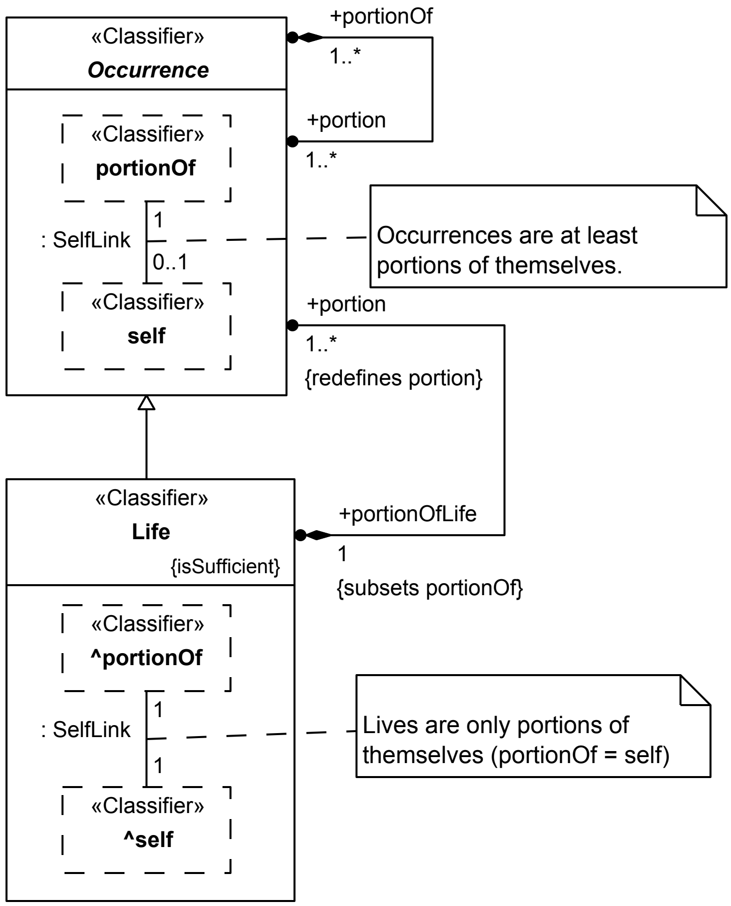


**Figure 36. Lives**

## 8.3.2 Elements

### 8.3.2.1 HappensBefore <Association>

**Description**

HappensBefore links an `earlier` Occurrence to a `later` one. The Occurrences do not overlap in time; none of their `snapshots` happen at the same time. This means no Occurrence HappensBefore itself.

**General Classes**

BinaryLink

**Attributes**

earlier : Occurrence {redefines sourceParticipant}

The earlier of the two participants in this HappensBefore Link.

later : Occurrence {redefines targetParticipant}

The later of the two participants in this HappensBefore Link.

**Constraints**

No constraints.

### 8.3.2.2 HappensDuring <Association>

**Description**

HappensDuring links a `shorter` Occurrence to a `longer` one. The `shorter` Occurrence completely overlaps the `longer` one in time; all `snapshots` of the `shorter` Occurrence happen at the same time as some `snapshot` of the `longer` one. This means every Occurrence HappensDuring itself.

**General Classes**

BinaryLink

**Attributes**

longer : Occurrence {redefines targetParticipant}

The longer of the two participants in this HappensDuring Link.

shorter : Occurrence {redefines sourceParticipant}

The shorter of the two participants in this HappensDuring Link.

**Constraints**

No constraints.

### 8.3.2.3 HappensWhile <Association>

**Description**

HappensWhile is a HappensDuring and its inverse. This means the linked Occurrences completely overlap each other in time (they happen at the same time); all `snapshots` of each Occurrence happen at the same time as one of the `snapshots` of other. This means every Occurrence HappensWhile itself.

**General Classes**

HappensDuring

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.3.2.4 Life <Classifier>

**Description**

Life is the class of Occurrences that are "maximal portions". That is, they are only portions of themselves.

**General Classes**

Occurrence

**Attributes**

portion : Occurrence [1..*] {redefines portion}

Occurrences that are portions of this Life, including at least this Life.

**Constraints**

No constraints.

### 8.3.2.5 Occurrence <Classifier>

**Description**

Occurrence is Anything that happens over time and space (the four physical dimensions). Other Occurrences can be portions of them within time and space, including slices in time, leading to snapshots that take zero time.

**General Classes**

Anything

**Attributes**

endShot : Occurrence {subsets snapshot}

`snapshot` that `happensAfter` all the others in this Occurrence, except possibly the `startShot` (when the Occurrence takes zero time).

endShotOf : Occurrence [0..*]

Inverse of `endShot` (Occurrences of which this is the `endShot`).

happensAfter : Occurrence [0..*] {subsets source}

Inverse of happensBefore (Occurrences that end when this one starts or earlier).

happensBefore : Occurrence [0..*] {subsets target}

Occurrences that start when this one ends or later.

happensDuring : Occurrence [0..*] {subsets target}

Occurrences that start when this one does or earlier and end when this one does or later (including this one).

happensDuring?[1] : Occurrence [0..*] {subsets source}

Inverse of happensDuring (Occurrences that start no earlier than this one and end no later, including this one).

happensWhile : Occurrence [0..*] {subsets happensDuring}

Occurrences that start and end at the same time as this one.

happensWhile?[1] : Occurrence [0..*] {subsets happensDuring?[1]}

Inverse of happensWhile (Occurrences that start and end at the same time as this one).

incomingTransfer : Transfer [0..*]

incomingTransferToSelf : Transfer [0..*] {subsets incomingTransfer}

Transfers for which this Occurrence is the `targetParticipant`.

outgoingTransfer : Transfer [0..*]

outgoingTransferFromSelf : Transfer [0..*] {subsets outgoingTransfer}

Transfers for which this Occurrence is the `sourceParticipant`.

portion : Occurrence [1..*] {subsets happensDuring?[1]}

Occurrences that happen within the time and space of this one (including this one) and that are considered the same thing occurring, see Life.

portionOf : Occurrence [1..*] {subsets happensDuring}

Inverse of `portion` (Occurrences within which this one happens in time and space, including this one, and that are considered the same thing occurring, see Life).

portionOfLife : Life {subsets portionOf}

The Life of which this Occurrence is a portion.

self : Occurrence {redefines self}

This Occurrence.

snapshot : Occurrence [1..*] {subsets timeSlice}

`timeSlices` of this Occurrence that take zero time.

snapshotOf : Occurrence [0..*]

Inverse of `snapshot` (Occurrences of which this is a `snapshot`).

startShot : Occurrence {subsets snapshot}

`snapshot` that `happensBefore` all the others in this Occurrence, except possibly the `endShot` (when the Occurrence takes zero time).

startShotOf : Occurrence [0..*]

Inverse of `startShot` (Occurrences of which this the `startShot`).

timeSlice : Occurrence [1..*] {subsets portion}

`portions` of this Occurrence that occupy its entire space during the entire portion, including this Occurrence.

timeSliceOf : Occurrence [1..*]

Inverse of `timeSlice` (Occurrences of which this is a `timeSlice` ).

transferBeforeTarget : Occurrence [0..*] {redefines transferTarget, happensBefore}

Occurrences whose input is the target of a TransferBefore of items from this Occurrence.

transferTarget : Occurrence [0..*] {subsets target}

Occurrences whose input is the target of a Transfer of items from this Occurrence.

**Constraints**

No constraints.

### 8.3.2.6 successions <Feature>

**Description**

`successions` is a specialization of `binaryLinks` restricted to type `HappensBefore`. It is the most general Succession (M1 instance of M2 Succession). All other Successions (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

HappensBefore
binaryLinks

**Attributes**

[no name] : Occurrence

[no name] : Occurrence

**Constraints**

No constraints.

## 8.4 Objects

## 8.4.1 Objects Overview

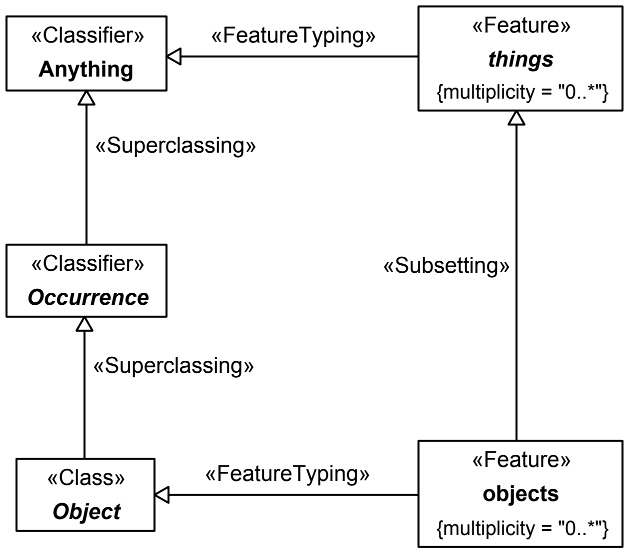


**Figure 37. Objects**

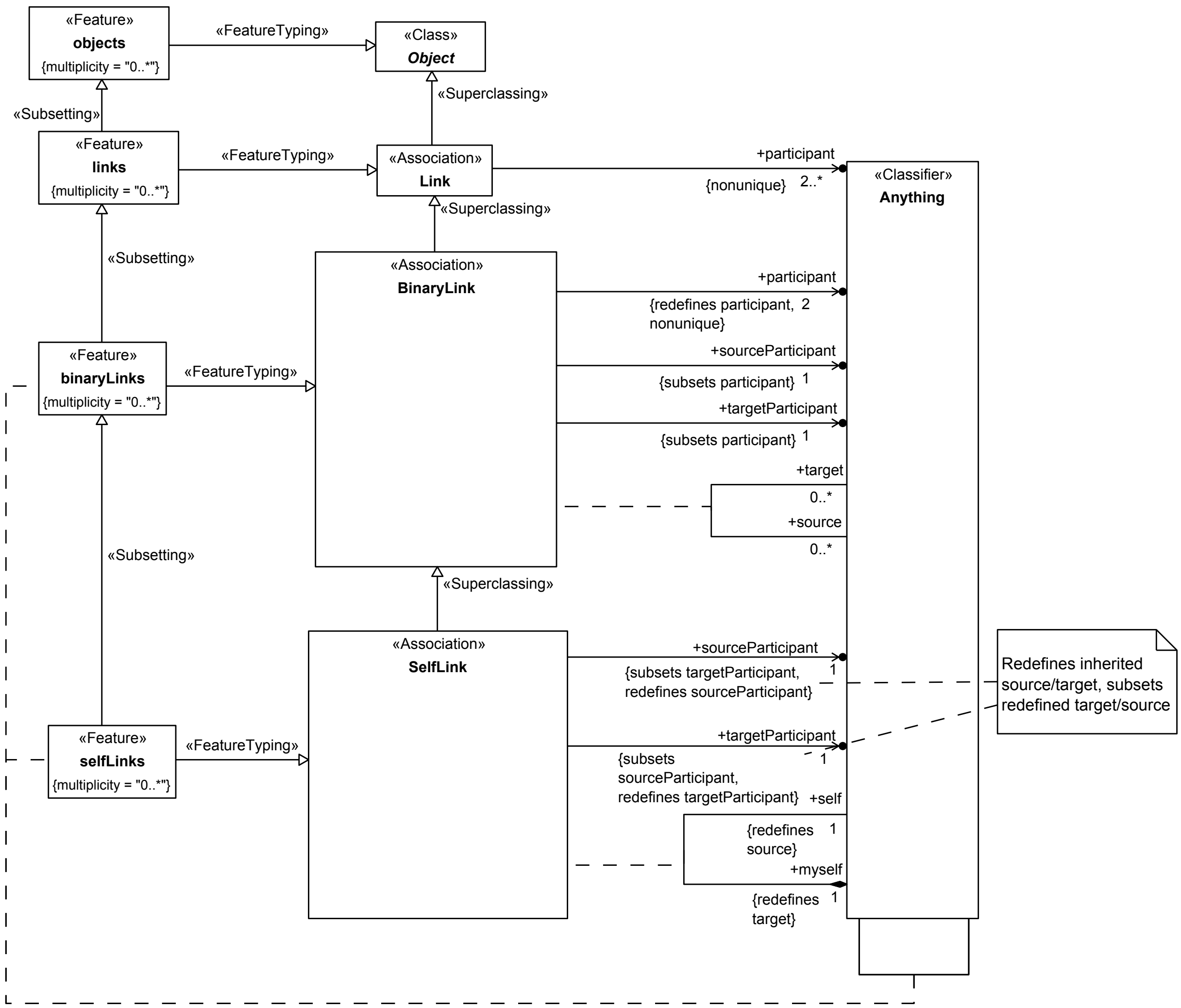


**Figure 38. Links**

## 8.4.2 Elements

### 8.4.2.1 BinaryLink <Association>

**Description**

BinaryLink is a Link with exactly two participant Features ("binary" Association). All other binary associations (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

Link

**Attributes**

participant : Anything {redefines participant, nonunique}

The participants of this BinaryLink, which are restricted to be exactly two.

source : Anything [0..*]

The end Feature of this BinaryLink corresponding to the `sourceParticipant`.

sourceParticipant : Anything {subsets participant}

The participant that is the source of this BinaryLink.

target : Anything [0..*]

The end Feature of this BinaryLink corresponding to the `targetParticipant`.

targetParticipant : Anything {subsets participant}

The participant that is the target of this BinaryLink.

**Constraints**

No constraints.

### 8.4.2.2 binaryLinks <Feature>

**Description**

`binaryLinks` is a specialization of `links` restricted to type BinaryLink. All other Features typed by BinaryLink or its specializations (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

links
BinaryLink

**Attributes**

[no name] : Anything

[no name] : Anything

**Constraints**

No constraints.

### 8.4.2.3 Link <Association>

**Description**

Link is an Object that is the most general Association (M1 instance of M2 Association). All other Associations (in libraries or user models) specialize it (directly or indirectly). Specializations of Link are domains of Features subsetting Link::participants, exactly as many as associationEnds of the Association classifying it, each with multiplicity 1. Values of Link::participants on specialized Links must be a value of at least one of its subsetting Features.

**General Classes**

Object

**Attributes**

participant : Anything [2..*] {nonunique}

The things linked by this Link.

**Constraints**

No constraints.

### 8.4.2.4 links <Feature>

**Description**

`links` is a specialization of `objects` restricted to type Link. It is the most general feature typed by Link. All other Features typed by Link or its specializations (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

objects
Link

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.4.2.5 Object <Class>

**Description**

Object is an Occurrence that is not a Performance. It is most general Class (M1 instance of M2 Class). All other Classes (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

Occurrence

**Attributes**

enactedPerformance : Performance [0..*] {subsets happensDuring?[1]}

Performances that are enacted by this object.

involvedIn : Performance [0..*]

Performances in which this Object is involved.

**Constraints**

No constraints.

### 8.4.2.6 objects <Feature>

**Description**

`objects` is a specialization of things restricted to type Object.

**General Classes**

Object
things

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.4.2.7 SelfLink <Association>

**Description**

SelfLink is a BinaryLink where the sourceParticipant and targetParticipant are the same. All other BinaryLinks where this is the case specialize it (directly of indirectly).

**General Classes**

BinaryLink

**Attributes**

myself : Anything {redefines target}

The association end corresponding to the `targetParticipant` of this SelfLink.

self : Anything {redefines source}

The association end corresponding to the `sourceParticipant` of this SelfLink.

sourceParticipant : Anything {subsets targetParticipant, redefines sourceParticipant}

The source participant of this SelfLink, which must be the same as the target participant.

targetParticipant : Anything {subsets sourceParticipant, redefines targetParticipant}

The target participant of this SelfLink, which must be the same as the source participant.

**Constraints**

No constraints.

### 8.4.2.8 selfLinks <Feature>

**Description**

`selfLinks` is a specialization of `binaryLinks` restricted to type SelfLink. It is the most general BindingConnector.  All other BindingConnectors (in libraries or user models) specialize it (directly or indirectly).

**General Classes**

SelfLink
binaryLinks

**Attributes**

[no name] : Anything

[no name] : Anything

**Constraints**

No constraints.

## 8.5 Performances

The Performances library provides elements to classify dynamic spans of time. Performances are the library base elements for to constrain the changes of objects over time. These support descriptions to generic or specific objects and also of objects alone or in interaction.

Evaluations are library base elements to constrain changes over time that result in the determination of specific values. They specialize into library elements for different kinds of expected results, such as LiteralEvaluation or BooleanEvaluation.

Specialized Link library elements are provided to indicate whether an object is intended to performing the given behavior or simply involved in it. The Performs relationship indicates that instances of Performance are "inside" their linked Object instances, indicating that the universe of affected states is expected to be those of the object. Performance instances linked by Involves instances to Object instances may describe object state evolution or reference dynamics outside those objects.

## 8.5.1 Performances Overview

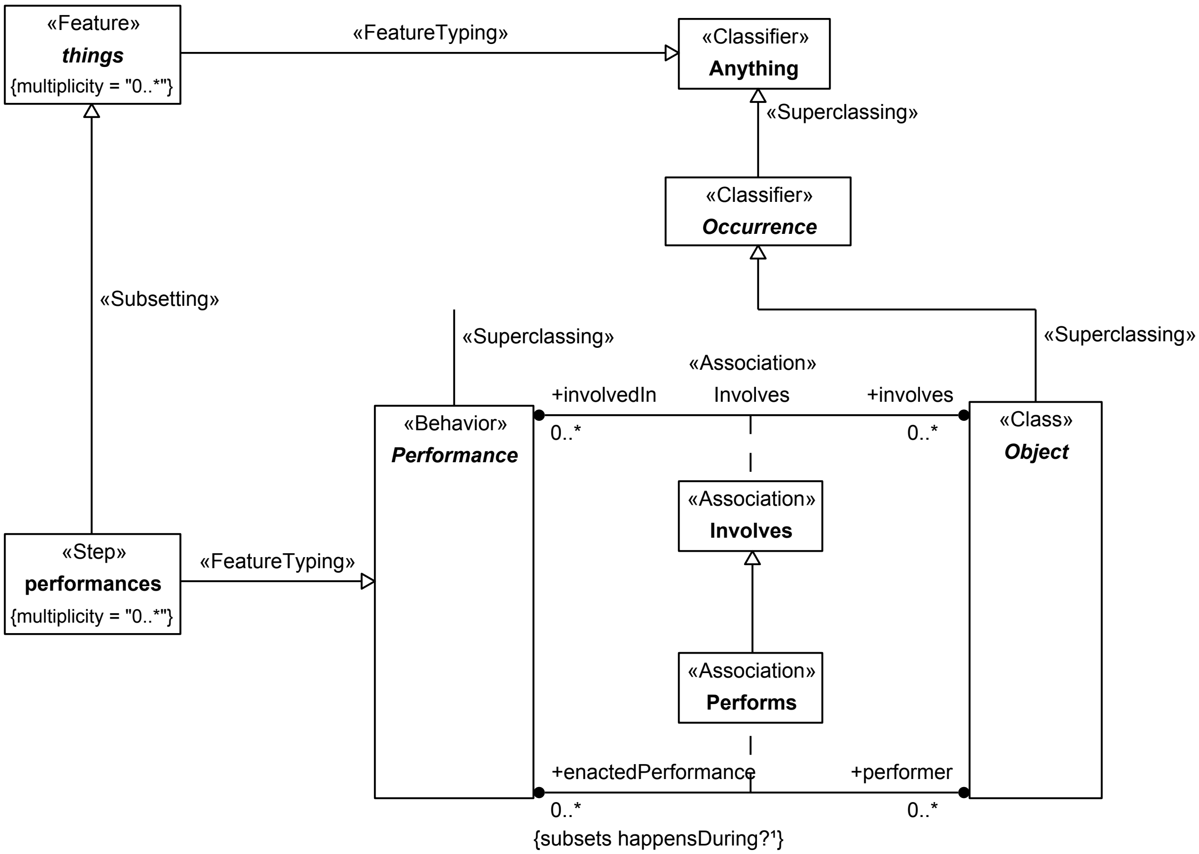


**Figure 39. Performances**

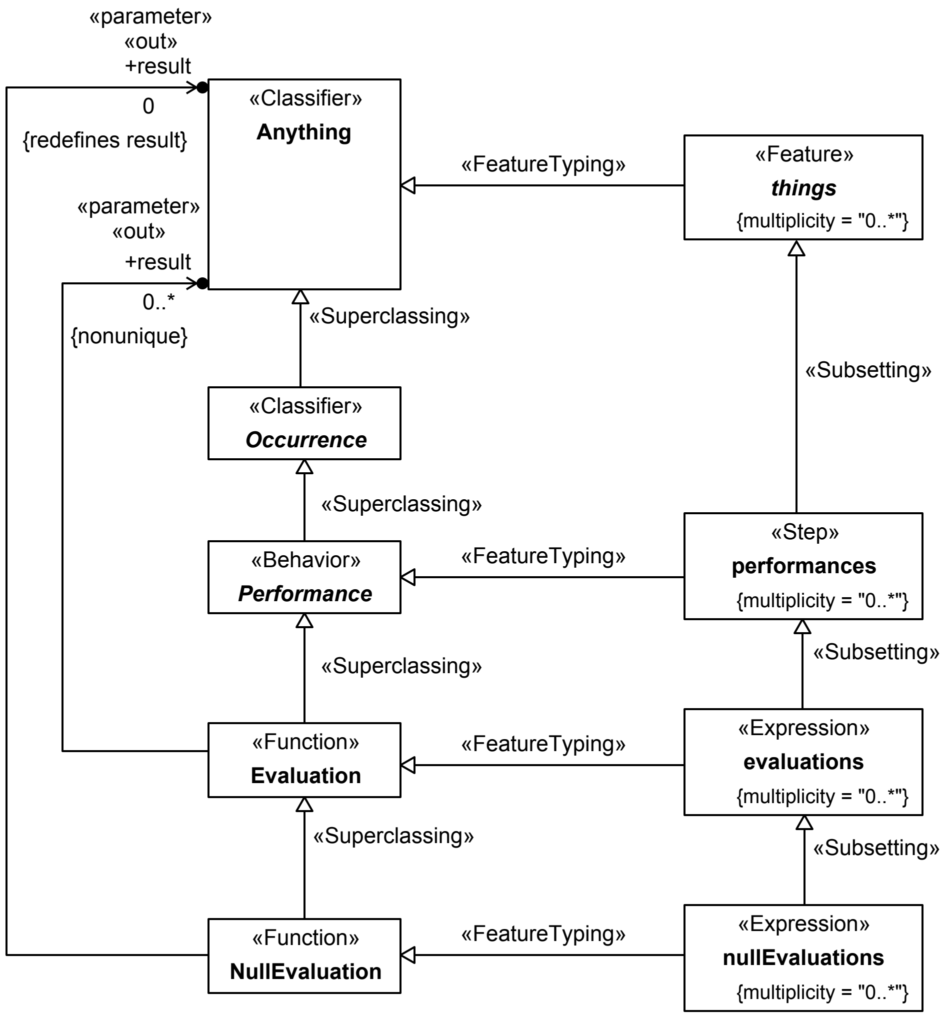


**Figure 40. Evaluations**

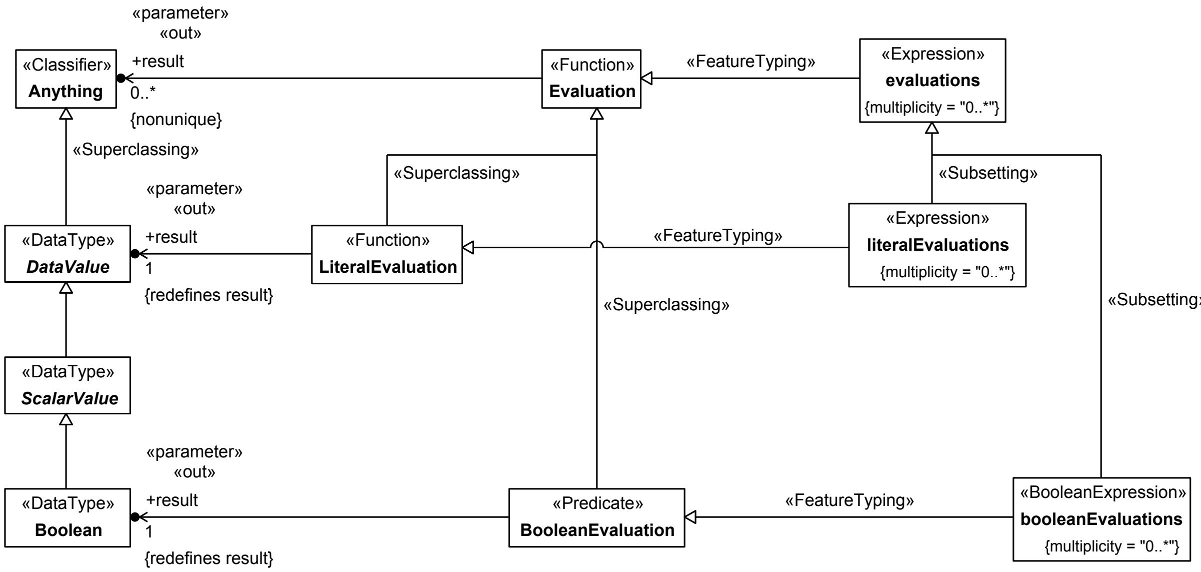


**Figure 41. Literal and Boolean Evaluations**

## 8.5.2 Elements

### 8.5.2.1 BooleanEvaluation <Predicate>

**Description**

BooleanEvaluation is a specialization of Evaluation that is the most general predicate that may be evaluated to produce a Boolean truth value.

**General Classes**

Evaluation

**Attributes**

result : Boolean {redefines result}

The Boolean result of this BooleanExpression.

**Constraints**

No constraints.

### 8.5.2.2 booleanEvaluations <BooleanExpression>

**Description**

`booleanEvaluations` is a specialization of `evaluations` restricted to type BooleanEvaluation.

**General Classes**

BooleanEvaluation
evaluations

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.5.2.3 Evaluation <Function>

**Description**

Evaluation is a Performance that ends with the production of a result.

**General Classes**

Performance

**Attributes**

result : Anything [0..*] {nonunique}

The `result` is the outcome of the Evaluation.

**Constraints**

No constraints.

### 8.5.2.4 evaluations <Feature>

**Description**

`evaluations` is a specialization of `performances` for Evaluations of functions.

**General Classes**

performances
Evaluation

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.5.2.5 Involves <Association>

**Description**

Involves classifies relationships between Performances and Objects.

**General Classes**

No general classes.

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.5.2.6 LiteralEvaluation <Function>

**Description**

LiteralEvaluation is a specialization of Evaluation for the case of LiteralExpressions.

**General Classes**

Evaluation

**Attributes**

result : DataValue {redefines result}

The result of this LiteralEvaluation, which is always a single DataValue.

**Constraints**

No constraints.

### 8.5.2.7 literalEvaluations <Expression>

**Description**

`literalEvaluations` is a specialization of `evaluations` restricted to type LiteralEvaluation.

**General Classes**

LiteralEvaluation
evaluations

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.5.2.8 NullEvaluation <Function>

**Description**

NullEvaluation is a specialization of Evaluation for the case of null expressions.

**General Classes**

Evaluation

**Attributes**

result : Anything {redefines result}

The result of this NullEvaluation, which always must be empty (i.e., "null").

**Constraints**

No constraints.

### 8.5.2.9 nullEvaluations <Expression>

**Description**

`evaluations` is a specialization of `performances` for Evaluations of functions.

**General Classes**

NullEvaluation
evaluations

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.5.2.10 Performance <Behavior>

**Description**

Performances classify spans of time and may apply constraints to how objects interact or change over those spans.

**General Classes**

Occurrence

**Attributes**

involves : Object [0..*]

Objects that are involved in this Performance.

performer : Object [0..*]

Objects that enact this performance.

**Constraints**

No constraints.

#### 8.5.2.11 performances <Feature>

**Description**

`performances` is the most general feature for Performances of behaviors.

**General Classes**

Performance
things

**Attributes**

No attributes.

**Constraints**

No constraints.

#### 8.5.2.12 Performs <Association>

**Description**

Performs is a specialization of Involves that asserts that the `performer` enacts the behavior carried out by the `enactedPerformance`.

**General Classes**

Involves

**Attributes**

No attributes.

**Constraints**

No constraints.

## 8.6 Transfers

## 8.6.1 Transfers Overview

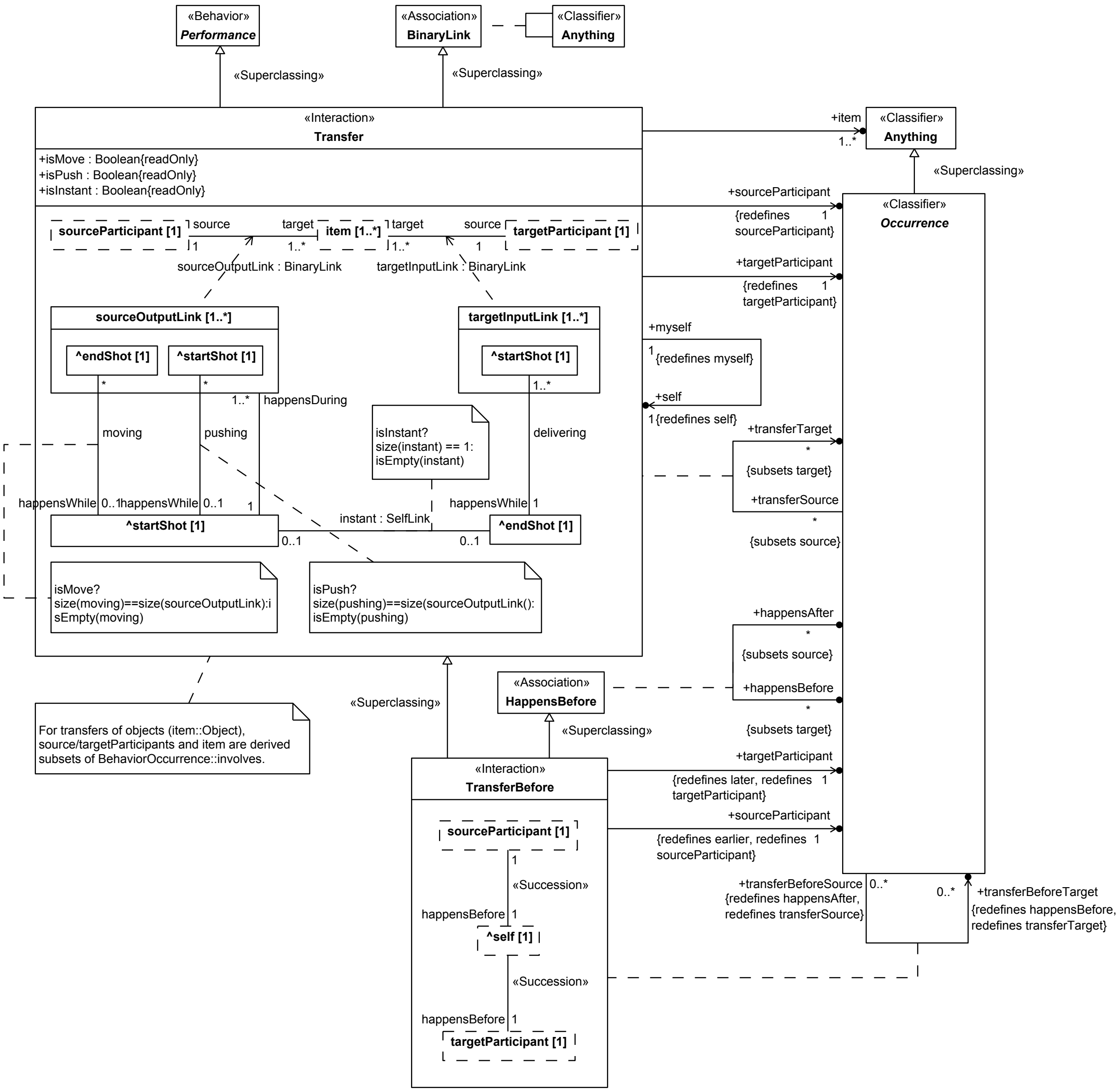


**Figure 42. Transfers**

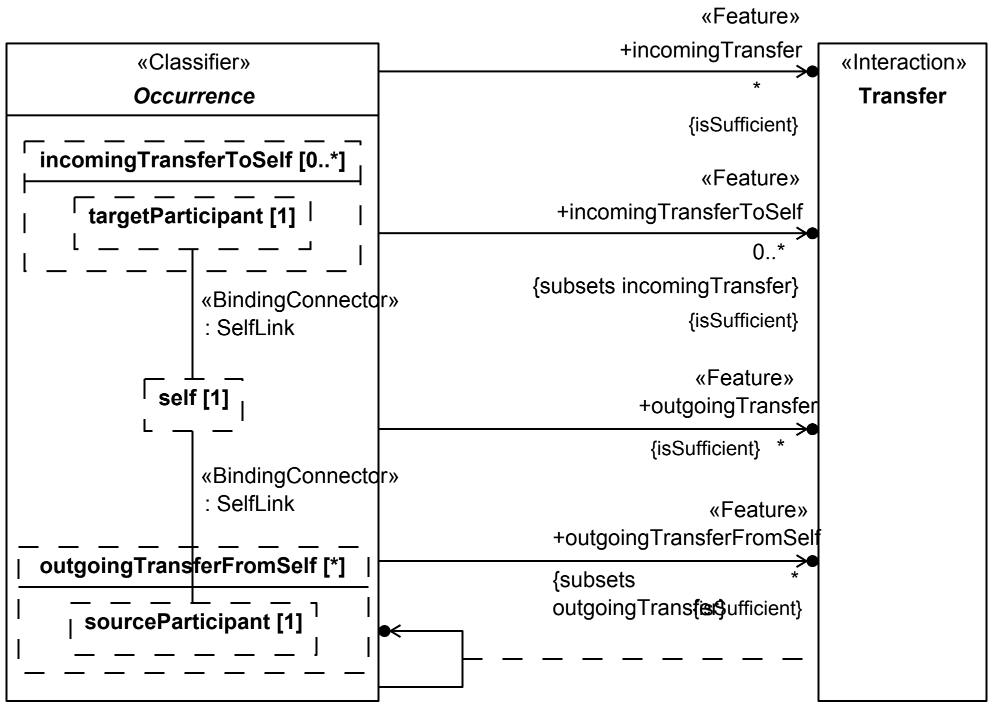


**Figure 43. Transfers, Incoming / Outgoing**

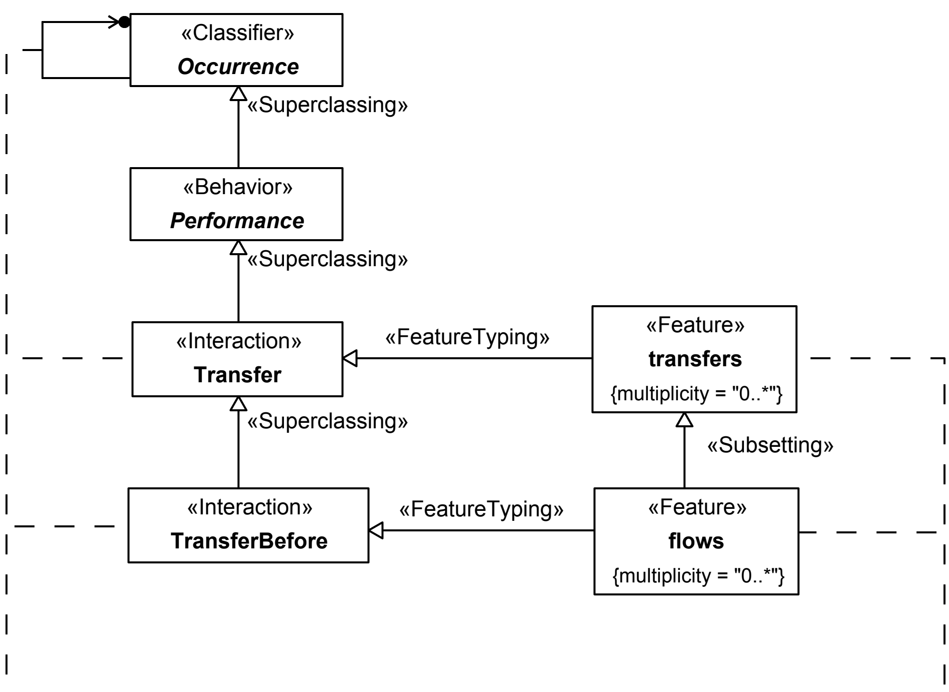


**Figure 44. Transfers, Features**

## 8.6.2 Elements

### 8.6.2.1 Transfer <Interaction>

**Description**

**General Classes**

Performance
BinaryLink

**Attributes**

isInstant : Boolean

isMove : Boolean

isPush : Boolean

item : Anything [1..*]

self : Transfer {redefines self}

sourceOutputLink : BinaryLink [1..*]

sourceParticipant : Occurrence {redefines sourceParticipant}

sourceSendShot : Occurrence

targetInputLink : BinaryLink [1..*]

targetParticipant : Occurrence {redefines targetParticipant}

targetReceiveShot : Occurrence

transferSource : Occurrence [0..*] {subsets source}

**Constraints**

No constraints.

### 8.6.2.2 transfers <Feature>

**Description**

**General Classes**

Transfer

**Attributes**

[no name] : Occurrence

[no name] : Occurrence

**Constraints**

No constraints.

#### 8.6.2.3 TransferBefore <Interaction>

**Description**

**General Classes**

Transfer
HappensBefore

**Attributes**

sourceParticipant : Occurrence {redefines earlier, sourceParticipant}

targetParticipant : Occurrence {redefines later, targetParticipant}

transferBeforeSource : Occurrence [0..*] {redefines happensAfter, transferSource}

**Constraints**

No constraints.

#### 8.6.2.4 transfersBefore <Feature>

**Description**

**General Classes**

TransferBefore
transfers

**Attributes**

[no name] : Occurrence

[no name] : Occurrence

**Constraints**

No constraints.

## 8.7 Control Performances

## 8.7.1 Control Performances Overview

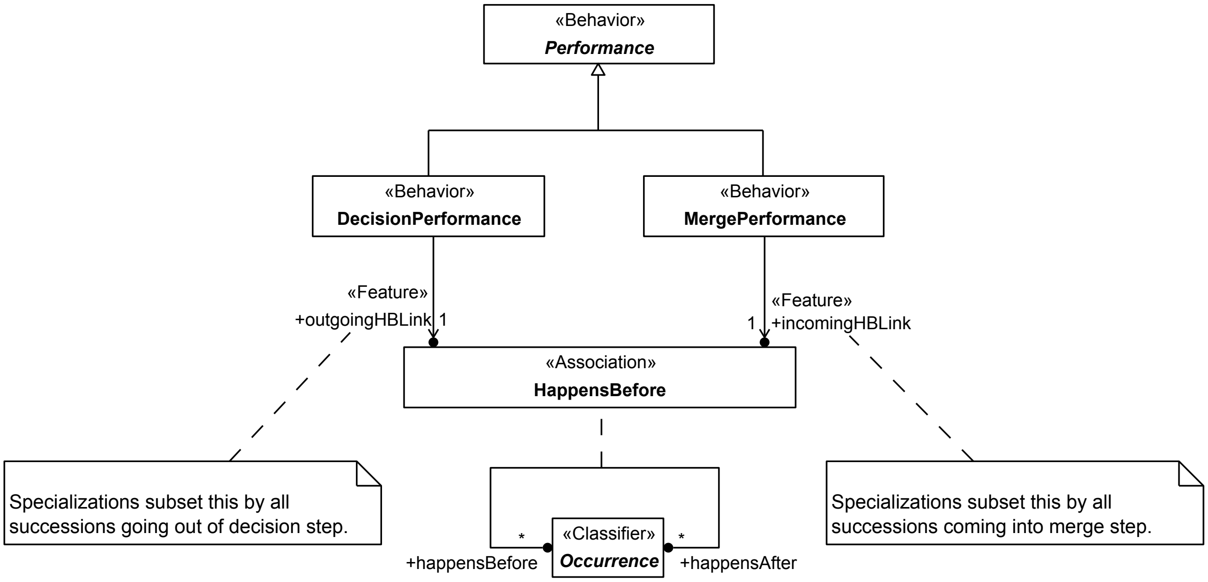


**Figure 45. Control Performances**

## 8.7.2 Elements

### 8.7.2.1 DecisionPerformance <Behavior>

**Description**

**General Classes**

Performance

**Attributes**

outgoingHBLink : HappensBefore

**Constraints**

No constraints.

### 8.7.2.2 MergePerformance <Behavior>

**Description**

**General Classes**

Performance

**Attributes**

incomingHBLink : HappensBefore

**Constraints**

No constraints.

# 8.8 State Performances

## 8.8.1 State Performances Overview

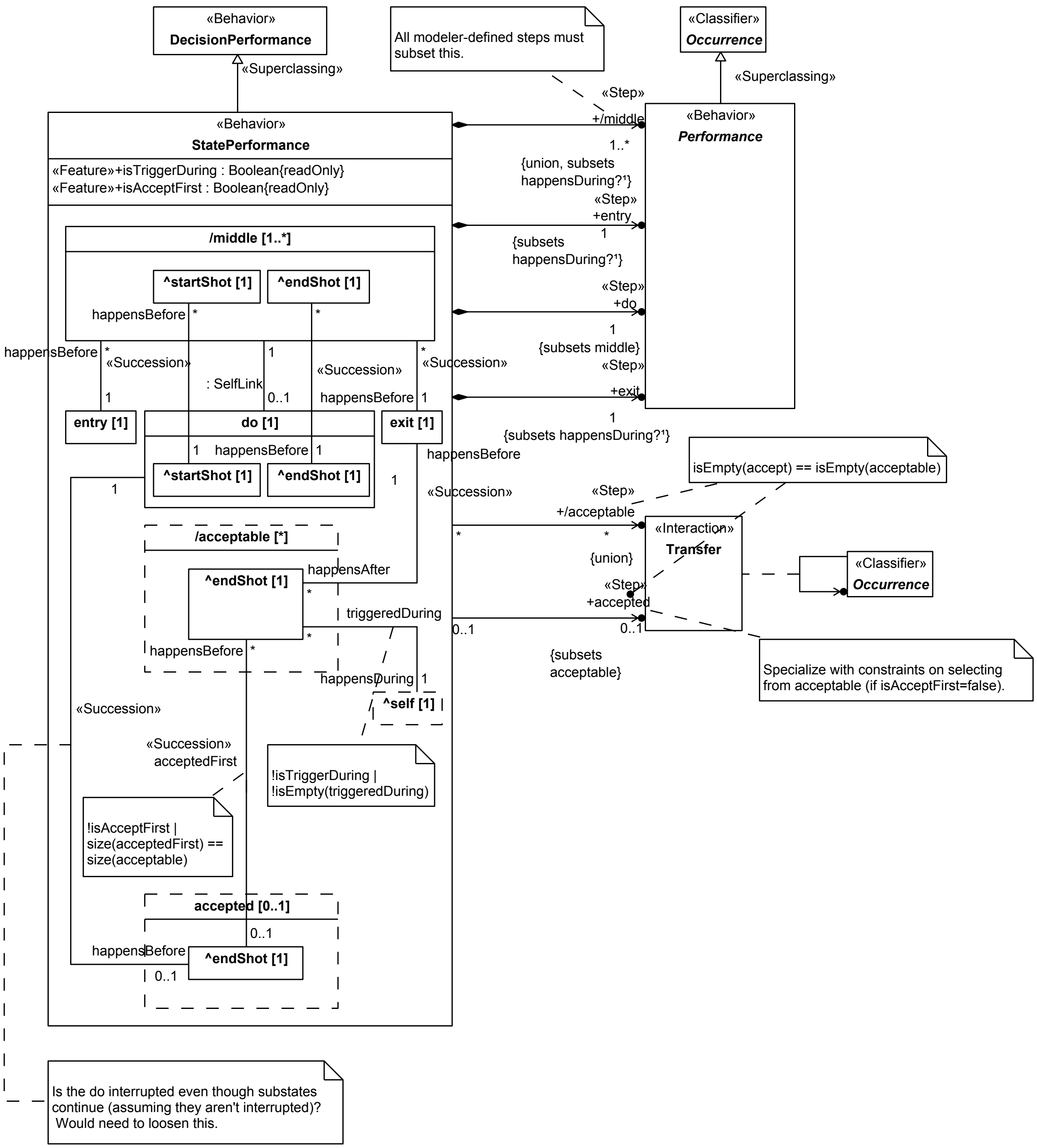


**Figure 46. State Performances**

## 8.8.2 Elements

### 8.8.2.1 StatePerformance <Behavior>

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.9 Transition Performances

## 8.9.1 Transition Performances Overview

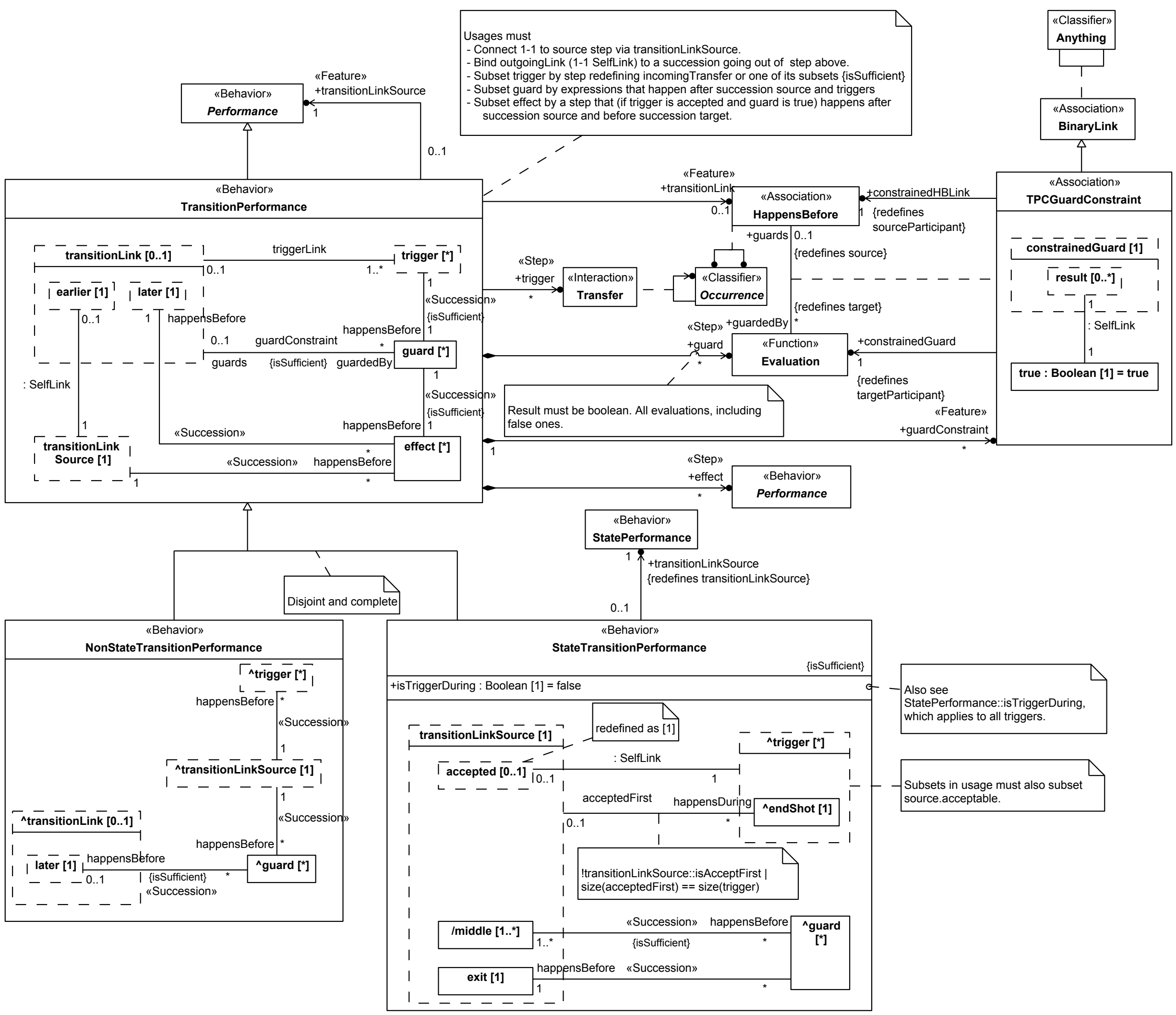


**Figure 47. Transition Performances**

## 8.9.2 Elements

### 8.9.2.1 TransitionPerformance <Behavior>

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.9.2.2 NonStateTransitionPerformance <>

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.9.2.3 StateTransitionPerformance <Behavior>

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.9.2.4 TPCGuardConstraint <Association>

**Description**

**General Classes**

BinaryLink

**Attributes**

constrainedGuard : Evaluation {redefines targetParticipant}

constrainedHBLink : HappensBefore {redefines sourceParticipant}

guardedBy : Evaluation [0..*] {redefines target}

guards : HappensBefore [0..1] {redefines source}

true : Boolean

**Constraints**

No constraints.

# 8.10 Scalar Values

## 8.10.1 Scalar Values Overview

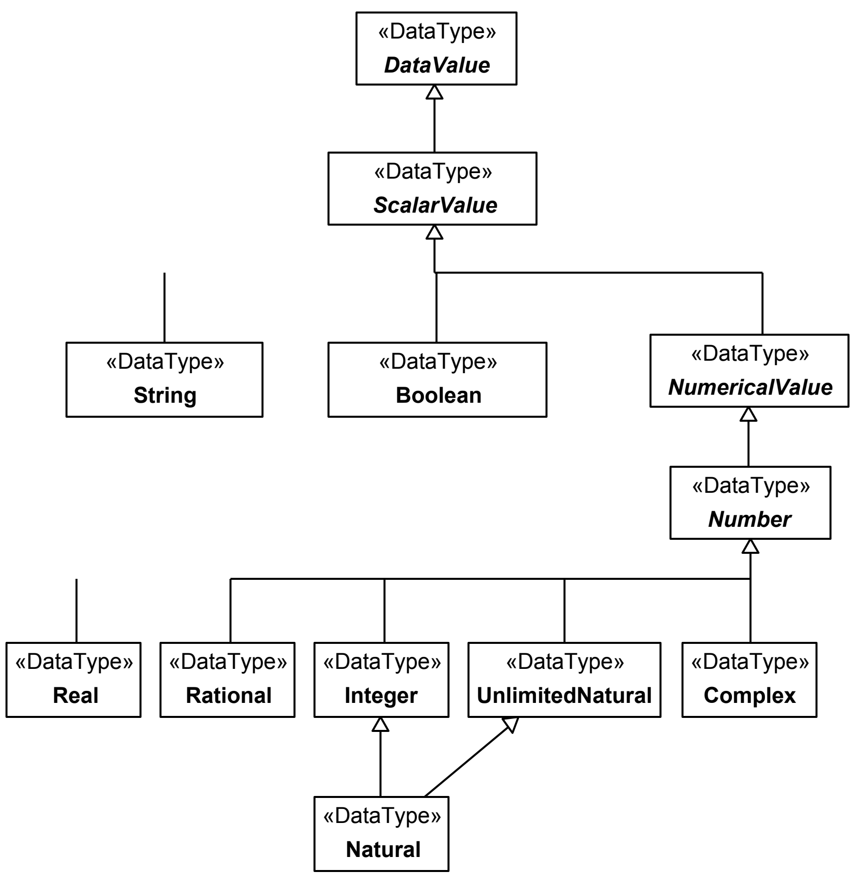


**Figure 48. Scalar Values**

## 8.10.2 Elements

#### 8.10.2.1 Booelan <DataType>

**Description**

**General Classes**

ScalarValue

**Attributes**

No attributes.

**Constraints**

No constraints.

#### 8.10.2.2 Complex <DataType>

**Description**

**General Classes**

Number

**Attributes**

No attributes.

**Constraints**

No constraints.

#### 8.10.2.3 Integer <DataType>

**Description**

**General Classes**

Number

**Attributes**

No attributes.

**Constraints**

No constraints.

#### 8.10.2.4 InternationalizedResourceIdentifer <DataType>

**Description**

**General Classes**

String

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.10.2.5 Natural <DataType>

**Description**

**General Classes**

DataValue
Integer
UnlimitedNatural

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.10.2.6 Number <DataType>

**Description**

**General Classes**

NumericalValue

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.10.2.7 NumericalValue <DataType>

**Description**

**General Classes**

ScalarValue

**Attributes**

No attributes.

**Constraints**

No constraints.

#### 8.10.2.8 Quaternion <DataType>

**Description**

**General Classes**

Number

**Attributes**

a : Real

b : Real

c : Real

d : Real

**Constraints**

No constraints.

#### 8.10.2.9 Rational <DataType>

**Description**

**General Classes**

Number

**Attributes**

No attributes.

**Constraints**

No constraints.

#### 8.10.2.10 Real <DataType>

**Description**

**General Classes**

Number

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.10.2.11 ScalarValue <DataType>

**Description**

**General Classes**

DataValue

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.10.2.12 String <DataType>

**Description**

**General Classes**

ScalarValue

**Attributes**

No attributes.

**Constraints**

No constraints.

### 8.10.2.13 UnlimitedNatural <DataType>

**Description**

**General Classes**

Number

**Attributes**

No attributes.

**Constraints**

No constraints.

## 8.11 Non-Scalar Values

## 8.11.1 Non-Scalar Values Overview

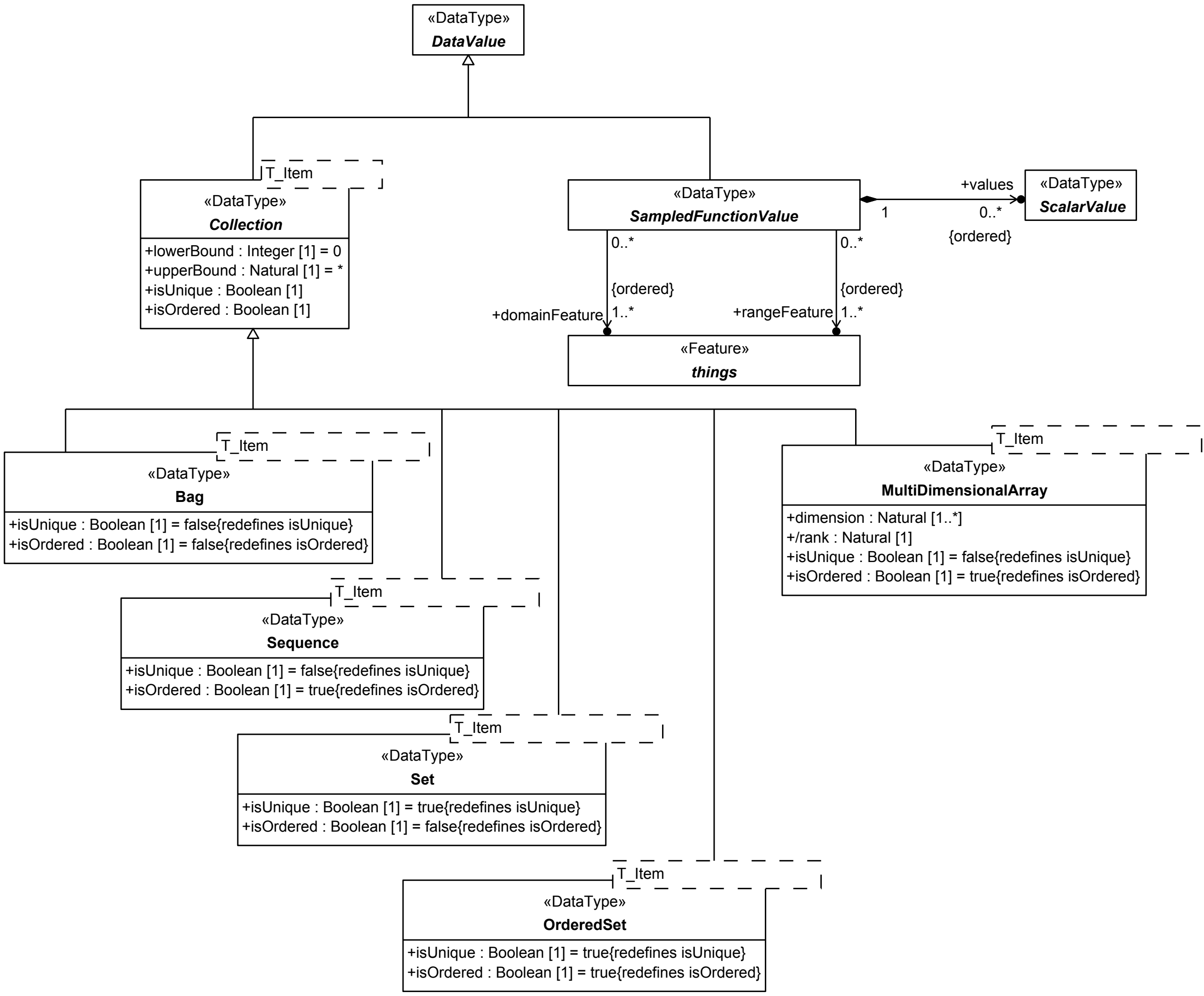


**Figure 49. Non-Scalar Values**

## 8.11.2 Elements

### 8.11.2.1 Bag <DataType>

**Description**

**General Classes**

Collection

**Attributes**

isOrdered : Boolean {redefines isOrdered}

isUnique : Boolean {redefines isUnique}

**Constraints**

No constraints.

#### 8.11.2.2 Collection <DataType>

**Description**

**General Classes**

DataValue

**Attributes**

isOrdered : Boolean

isUnique : Boolean

lowerBound : Integer

upperBound : Natural

**Constraints**

No constraints.

#### 8.11.2.3 MultiDimensionalArray <DataType>

**Description**

**General Classes**

Collection

**Attributes**

dimension : Natural [1..*]

isOrdered : Boolean {redefines isOrdered}

isUnique : Boolean {redefines isUnique}

/rank : Natural

**Constraints**

No constraints.

#### 8.11.2.4 OrderedSet <DataType>

**Description**

**General Classes**

Collection

**Attributes**

isOrdered : Boolean {redefines isOrdered}

isUnique : Boolean {redefines isUnique}

**Constraints**

No constraints.

#### 8.11.2.5 SampledFunctionValue <DataType>

**Description**

**General Classes**

DataValue

**Attributes**

domainFeature : things [1..*] {ordered}

rangeFeature : things [1..*] {ordered}

values : ScalarValue [0..*] {ordered}

**Constraints**

No constraints.

#### 8.11.2.6 Sequence <DataType>

**Description**

**General Classes**

Collection

**Attributes**

isOrdered : Boolean {redefines isOrdered}

isUnique : Boolean {redefines isUnique}

**Constraints**

No constraints.

#### 8.11.2.7 Set <DataType>

**Description**

**General Classes**

Collection

**Attributes**

isOrdered : Boolean {redefines isOrdered}

isUnique : Boolean {redefines isUnique}

**Constraints**

No constraints.

# 8.12 Base Functions

## 8.12.1 Base Functions Overview

## 8.12.2 Elements

### 8.12.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.13 Scalar Functions

## 8.13.1 Scalar Functions Overview

## 8.13.2 Elements

### 8.13.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.14 Boolean Functions

## 8.14.1 Boolean Functions Overview

## 8.14.2 Elements

### 8.14.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.15 String Functions

## 8.15.1 String Functions Overview

## 8.15.2 Elements

### 8.15.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.16 Numerical Functions

## 8.16.1 Numerical Functions Overview

## 8.16.2 Elements

### 8.16.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.17 Natural Functions

## 8.17.1 Natural Functions Overview

## 8.17.2 Elements

### 8.17.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.18 Integer Functions

## 8.18.1 Integer Functions Overview

## 8.18.2 Elements

### 8.18.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.19 UnlimitedNatural Functions

## 8.19.1 UnlimitedNatural Functions Overview

## 8.19.2 Elements

#### 8.19.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

## 8.20 Rational Functions

### 8.20.1 Rational Functions Overview

### 8.20.2 Elements

#### 8.20.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

## 8.21 Real Functions

### 8.21.1 Real Functions Overview

### 8.21.2 Elements

#### 8.21.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.22 Complex Functions

## 8.22.1 Complex Functions Overview

## 8.22.2 Elements

### 8.22.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.23 Non-Scalar Functions

## 8.23.1 NonScalar Functions Overview

## 8.23.2 Elements

### 8.23.2.1 Library Element

**Description**

**General Classes**

Model Library Element Description

**Attributes**

No attributes.

**Constraints**

No constraints.

# 8.24 Control Functions

## 8.24.1 Control Functions Overview

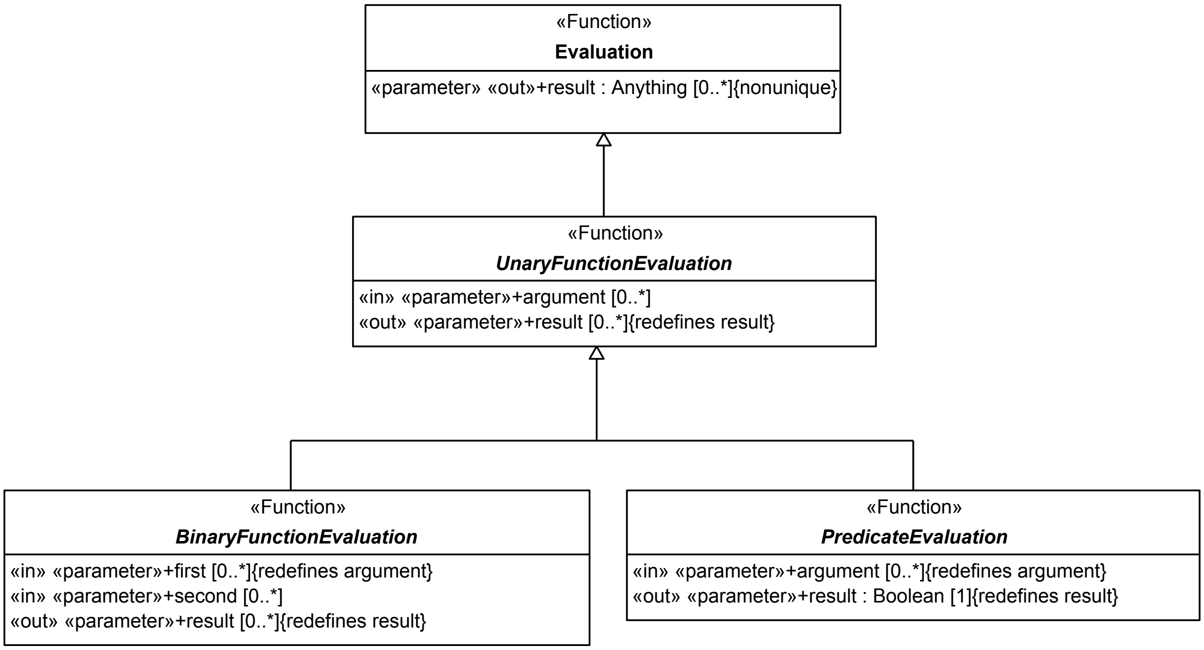


**Figure 50. Operation Functions**

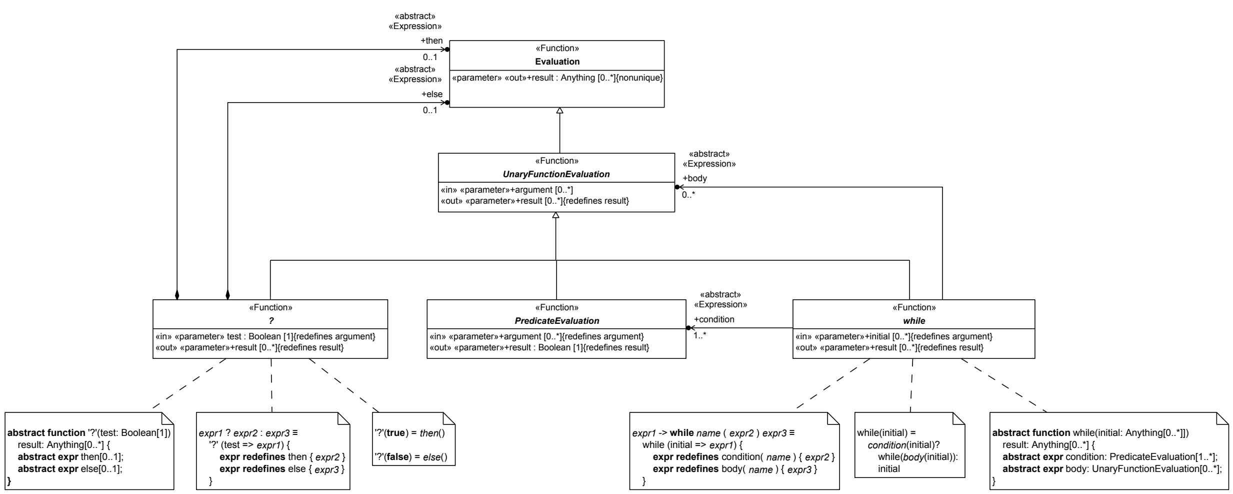


**Figure 51. Conditional Control Functions**

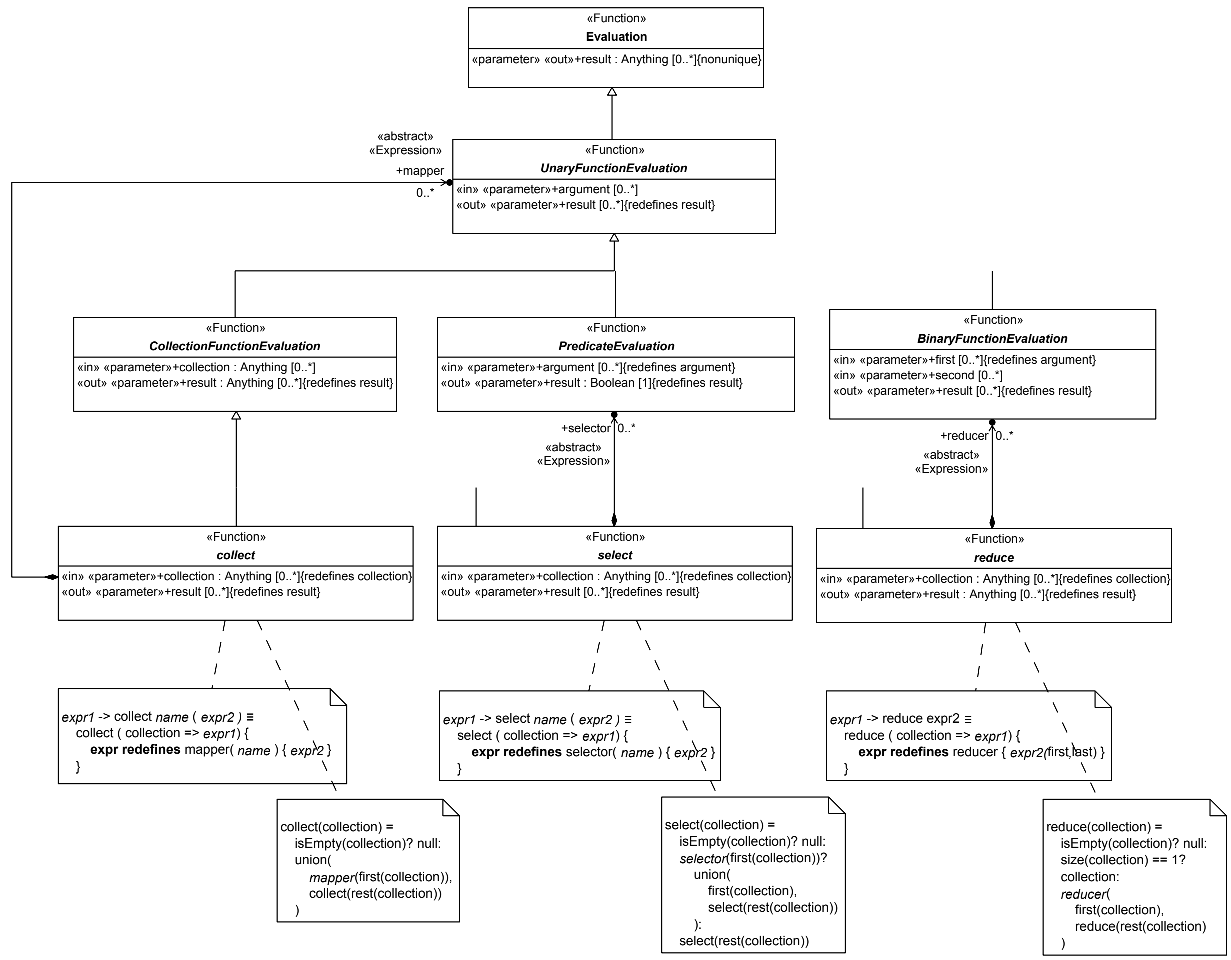


**Figure 52. Collection Control Functions**

## 8.24.2 Elements

### 8.24.2.1 ? <Function>

**Description**

**General Classes**

UnaryFunctionEvaluation

**Attributes**

else : Evaluation [0..1]

result [0..*] {redefines result}

test : Boolean {redefines argument}

then : Evaluation [0..1]

**Constraints**

No constraints.

#### 8.24.2.2 BinaryFunctionEvaluation <Function>

**Description**

**General Classes**

UnaryFunctionEvaluation

**Attributes**

first [0..*] {redefines argument}

result [0..*] {redefines result}

second [0..*]

**Constraints**

No constraints.

#### 8.24.2.3 Collect <Function>

**Description**

**General Classes**

CollectionFunctionEvaluation

**Attributes**

collection : Anything [0..*] {redefines collection}

mapper : UnaryFunctionEvaluation [0..*]

result [0..*] {redefines result}

**Constraints**

No constraints.

#### 8.24.2.4 CollectionFunctionEvaluation <Function>

**Description**

**General Classes**

UnaryFunctionEvaluation

**Attributes**

collection : Anything [0..*]

result : Anything [0..*] {redefines result}

**Constraints**

No constraints.

#### 8.24.2.5 PredicateEvaluation <Function>

**Description**

**General Classes**

UnaryFunctionEvaluation

**Attributes**

argument [0..*] {redefines argument}

result : Boolean {redefines result}

**Constraints**

No constraints.

#### 8.24.2.6 Reduce <Function>

**Description**

**General Classes**

CollectionFunctionEvaluation

**Attributes**

collection : Anything [0..*] {redefines collection}

reducer : BinaryFunctionEvaluation [0..*]

result : Anything [0..*] {redefines result}

**Constraints**

No constraints.

#### 8.24.2.7 Select <Function>

**Description**

**General Classes**

CollectionFunctionEvaluation

**Attributes**

collection : Anything [0..*] {redefines collection}

result [0..*] {redefines result}

selector : PredicateEvaluation [0..*]

**Constraints**

No constraints.

### 8.24.2.8 UnaryFunctionEvaluation <Function>

**Description**

**General Classes**

Evaluation

**Attributes**

argument [0..*]

result [0..*] {redefines result}

**Constraints**

No constraints.

### 8.24.2.9 While <Function>

**Description**

**General Classes**

UnaryFunctionEvaluation

**Attributes**

body : UnaryFunctionEvaluation [0..*]

condition : PredicateEvaluation [1..*]

initial [0..*] {redefines argument}

result [0..*] {redefines result}

**Constraints**

No constraints.

# 9 Model Interchange

KerML models may be interchanged between conformant KerML modeling tools (see Clause 2) using text files in any of the following formats:

1. Textual notation, using the textual concrete syntax defined in this specification. Note that in certain limited cases, models conformant with the KerML syntax, but prepared by a means other than using the KerML textual concrete syntax, may not be fully serializable into the standard textual notation. In this case, a tool may either not export such model at all using the textual notation, or export the model as closely as possible, informing the user of any changes from the original model.
2. JSON, using a format consistent with the JSON schema based on the KerML abstract syntax, consistent with the REST/HTTP platform-specific binding of the Element Navigation Service of the Systems Modeling API and Services specification [SysAPI].
3. XML, using the XML Metadata Interchange [XMI] format based on the MOF-conformant abstract syntax metamodel for KerML.

Every conformant KerML modeling tool shall provide the ability to import and/or export (as appropriate) models in at least one of the first two formats.

**Submission Note.** Model interchange will be addressed more fully in the revised submission. Issues to be addressed include interchanging tool-generated metadata (such as Element identifiers) in the textual notation and full documentation of the JSON format.

# A Annex: Conformance Test Suite

**Submission Note.** The conformance test suite will be provided in the revised submission.