Date: May 2022

Kernel Modeling Language (KerML)

Version 1.0
Release 2022-04

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

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

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 are tailored to specific applications while maintaining fundamental semantic interoperability.

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 Partners

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. This is now publicly available under an open source license at https://github.com/Systems-Modeling. The latest release is the 2021-10 version, based on the KerML metamodel baselined in October 2021 and used for this submission. However, since the conformance test suite has not been developed as of the time of this submission, it is not possible to formally demonstrate the conformance of the implementation to the proposed specification. Nevertheless, the majority of this proposed specification describes the language as it has been implemented. For those specific areas in which the pilot implementation as of the 2021-10 release is known to not fully conform to the initial submission of the SysML specification, the deviations are identified in "implementation notes" in this document. The SST is planning to continue the incremental development of the pilot implementation and complete it for the final submission.

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 interchanged 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 built 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

[fUML] *Semantics of a Foundational Subset for Executable UML Models (fUML),* Version 1.4
https://www.omg.org/spec/fUML/1.4

[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. Test specifications cover common situations of system operation, but usually cannot cover all of them.

In the terms above, this specification serves as requirements for KerML language tooling, which 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 leads to two kinds of 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 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 by 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.

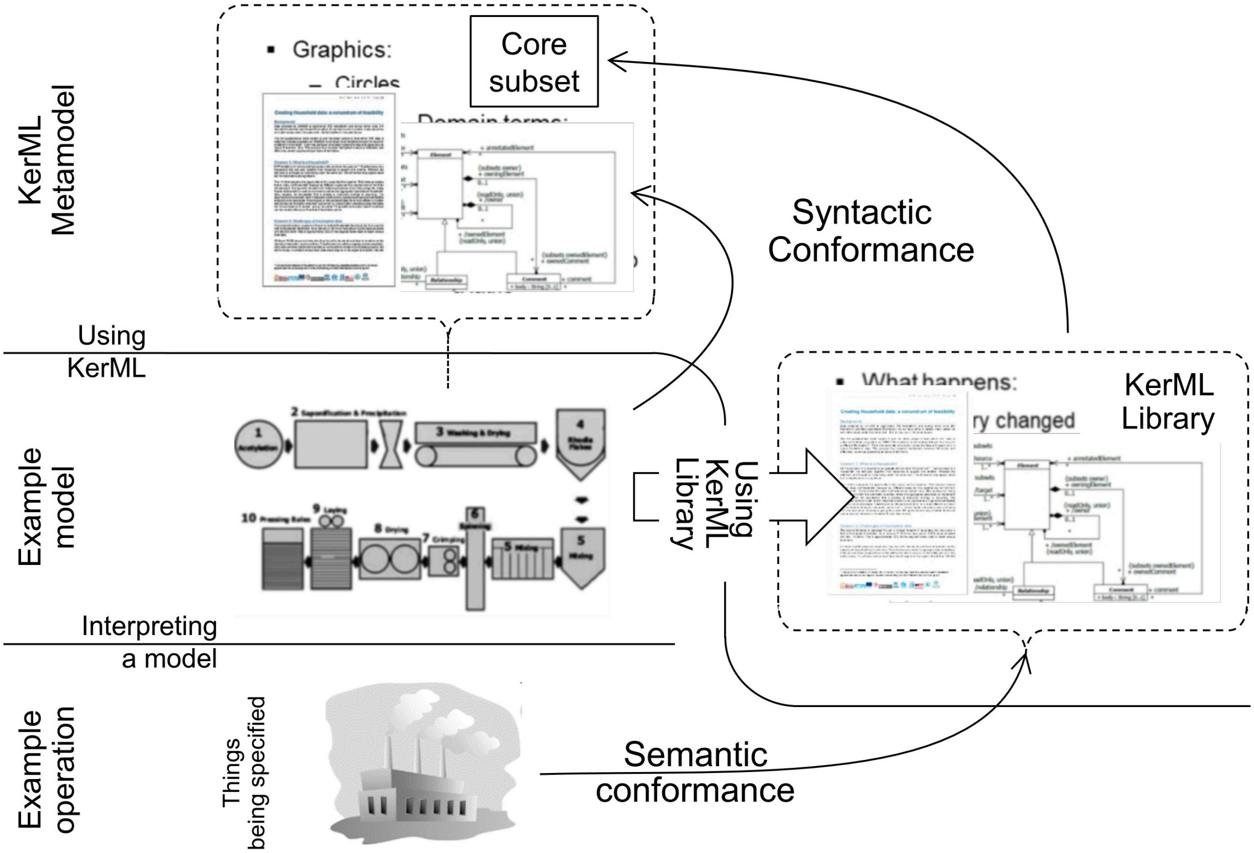

Figure 1. Syntactic and Semantic Conformance

6.2 Document 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 successive layers of the metamodel. The subclause 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 no 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 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).

Submission Note. Consideration will be given to re-organizing this document for the final submission to move overview and descriptive material out of the Metamodel clause into a separate Language Description clause, similarly to how the SysML specification is now organized.

6.3 Document Conventions

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

1. Names of metaclasses from the KerML abstract syntax model appear exactly as in the abstract syntax, including capitalization, except possibly with added pluralization. When used as English common nouns, e.g., "an Element", "multiple FeatureTypings", they refer to instances of the metaclass (in models). e.g., "Elements can own other Elements" refers to instances of the metaclass Element that reside in models. This can be modified with the term "metaclass" as necessary to refer to the metaclass itself instead of its instances, e.g., "The Element metaclass is contained in the Elements package."
2. Names of properties of metaclasses appear in "`code`" font. When used as English common nouns, e.g., “an `ownedRelatedElement`”, “multiple `featuringTypes`”, they refer to values of the properties. This can be modified using the term "metaproperty" as necessary to refer to the metaproperty itself instead of its values, e.g., "The `ownedRelatedElement` metaproperty is contained in the Elements package."

The following stylistic conventions apply to text about KerML models, including models in the Model Library (Clause 8):

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

(see 7.3 about instances (interpretations) of KerML Types and Features) In addition, KerML model elements appear in italicized font, including elements from the KerML Model Libraries (e.g., "*Behavior*" and "*`performances`*") and elements of sample user models (e.g., "*Vehicle*" and "*`wheels`*").

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. When individual keywords are referenced, they 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 first order logic notation, extended as follows:

1. Quantifiers can specify that variable values must be members of particular sets, 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 $\forall t_g, t_s\ t_g \in V_T \land t_s \in V_T \Rightarrow$ ...).
2. Dots (.) appearing between metaproperty names have the same meaning as in OCL, including implicit collections [OCL].
3. Sets are identified in the usual set-builder notation, which specifies members of a set between curly braces ("{}"). The notation is extended with "#" before an opening brace to refer to the cardinality of a set.

Element names appearing in the mathematical semantics refer to the element itself, rather than its instances, using the same font conventions as above. Mathematical terms used in the specification are defined in 7.3.1.2.

Release Note. A paragraph marked as a "release note" (like this one) is not to be considered part of the formal specification being proposed. Rather, it is a note describing either material that was not included at the time of this release of the proposed specification, or changes to the specification that are expected before the final submission of the proposal. Such notes will be removed in the final submission as the issues they address are resolved.

Implementation Note. A paragraph marked as an "implementation note" (like this one) is also not to be considered part of the formal specification being proposed. Rather, it describes an area in which the proof-of-concept pilot implementation being developed by the submission team is not fully consistent with what is being proposed in the specification as of the time of this submission. These notes will also be removed in the final submission.

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, Jet Propulsion Laboratory
- Bjorn Cole, Lockheed Martin Corporation

The primary authors of this specification document and the syntactic and library 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 Partners LLC

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 (v1 to v2 transformation 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 (v1 to v2 transformation 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 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 linguisitic 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 as associations and behavior, see 7.4. Its additional semantics are 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 =
    implementation defined character sequence
LINE_TEXT =
    character sequence excluding LINE_TERMINATORs
WHITE_SPACE =
    space | tab | form_feed | 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 =
    '//' LINE_TEXT

MULTILINE_NOTE =
    '//*' COMMENT_TEXT '*/'

REGULAR_COMMENT =
    '/*' NON_STAR_CHARACTER COMMENT_TEXT '*/'

PREFIX_COMMENT =
    '/**' COMMENT_TEXT '*/'

COMMENT_TEXT =
    ( COMMENT_LINE_TEXT | LINE_TERMINATOR )*

COMMENT_LINE_TEXT =
    LINE_TEXT excluding the sequence '*/'

NON_START_CHARACTER =
    any character other than '*'
```

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. *Prefix comment text* includes all the text from the initial characters "/**" to the final characters "*/". Document comment text is used for comments that are automatically about the lexically next element (see 7.2.3).

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

7.1.2.4 Names

```
NAME =
    BASIC_NAME | UNRESTRICTED_NAME

BASIC_NAME =
    BASIC_INITIAL_CHARACTER BASIC_NAME_CHARACTER*

UNRESTRICTED_NAME =
    single_quote ( NAME_CHARACTER | ESCAPE_SEQUENCE )* single_quote

BASIC_INITIAL_CHARACTER =
    ALPHABETIC_CHARACTER | '_'

BASIC_NAME_CHARACTER =
    BASIC_INITIAL_CHARACTER | DECIMAL_DIGIT

ALPHABETIC_CHARACTER =
    any character 'a' through 'z' or 'A' through 'Z'

DECIMAL_DIGIT =
    any character '0' through '9'

NAME_CHARACTER =
    any printable character other than backslash or single_quote

ESCAPE_SEQUENCE =
    see Table 1
```

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 =
    DECIMAL_DIGIT+

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

A *decimal value* represents 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 7.4.8.2.4). 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 7.4.8.2.4). 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
```

Release Note. The final submission may allow numeric literals other than decimal, particularly the traditional binary, octal and hexadecimal.

7.1.2.6 String Values

```
STRING_VALUE =
    '"' ( STRING_CHARACTER | ESCAPE_SEQUENCE )* '"'

STRING_CHARACTER =
    any printable character other than backslash or '"'
```

A *string value* lexically delimits a sequence of characters to be included in a String literal value (see 7.4.8). The characters in the string value are surrounded by double quotes, within which escape characters resolve to their meaning as given in Table 1. 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 of a basic name but cannot actually be used as a basic name. The following keywords are so reserved in KerML.

```
about abstract alias all and as assign assoc behavior binding bool by chains
class classifier comment composite conjugate conjugates conjugation connector
datatype default derived disjoining disjoint doc element else end expr false
feature featured featuring filter first flow for from function generalization
hastype id if implies import in inout interaction inv istype language member
metaclass metadata multiplicity namespace nonunique not null of or ordered
out package portion predicate private protected public readonly redefines
redefinition relationship rep return specialization specializes step struct
subclassifier subset subsets subtype succession then to true type typed
typing xor
```

7.1.2.8 Symbols

The *symbols* shown below are non-name tokens composed entirely of characters that are not alphanumeric. In some cases these symbols have no meaning themselves, but are used to allow unambiguous separation between other tokens that do have meaning. In other cases, they are distinguished notations in the KerML Expression sublanguage (see 7.4.8) that map to particular library Functions or symbolic shorthand for meaningful relationships.

```
( ) { } [ ] ; , ! != % & && * ** + - -> .. / : ::
:> :>> < <= = := == => > >= ? ?? @ ^ ^^ | || ~
```

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 keywords in the concrete syntax have an equivalent symbolic representation. For convenience, the concrete syntax grammar uses the following special lexical terminals, which match either the symbol or the corresponding keyword.

```
TYPED_BY    = ':'   | 'typed' 'by'
SPECIALIZES = ':>'  | 'specializes'
SUBSETS     = ':>'  | 'subsets'
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 Backus Naur Form (EBNF) notation (see Table 2) that includes further notations to describe how the concrete syntax maps to the abstract syntax (see Table 3).

Productions in the grammar formally result in the synthesis of classes in the abstract syntax and the population of their properties (see Table 4). 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 2. 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 3. 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 }`
	`{ x.p = value }`
	`{ x.p += value }`
		Assign (or add) the given `value` to the variable `v` or property `x.p`, without parsing any input.

Name resolution	`[QualifiedName]`	Parse a `QualifiedName`, then resolve that name to an Element reference (see 7.2.4.2.4) for use as a value in an assignment as above.

Table 4. 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.

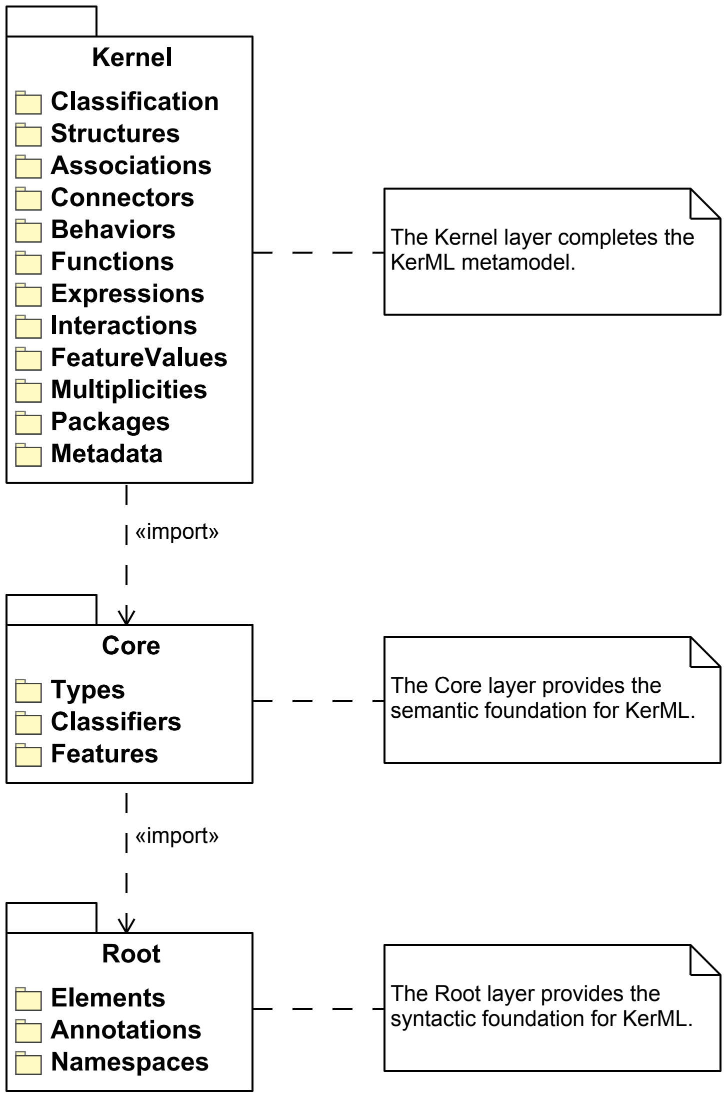

Figure 2. KerML Syntax Layers

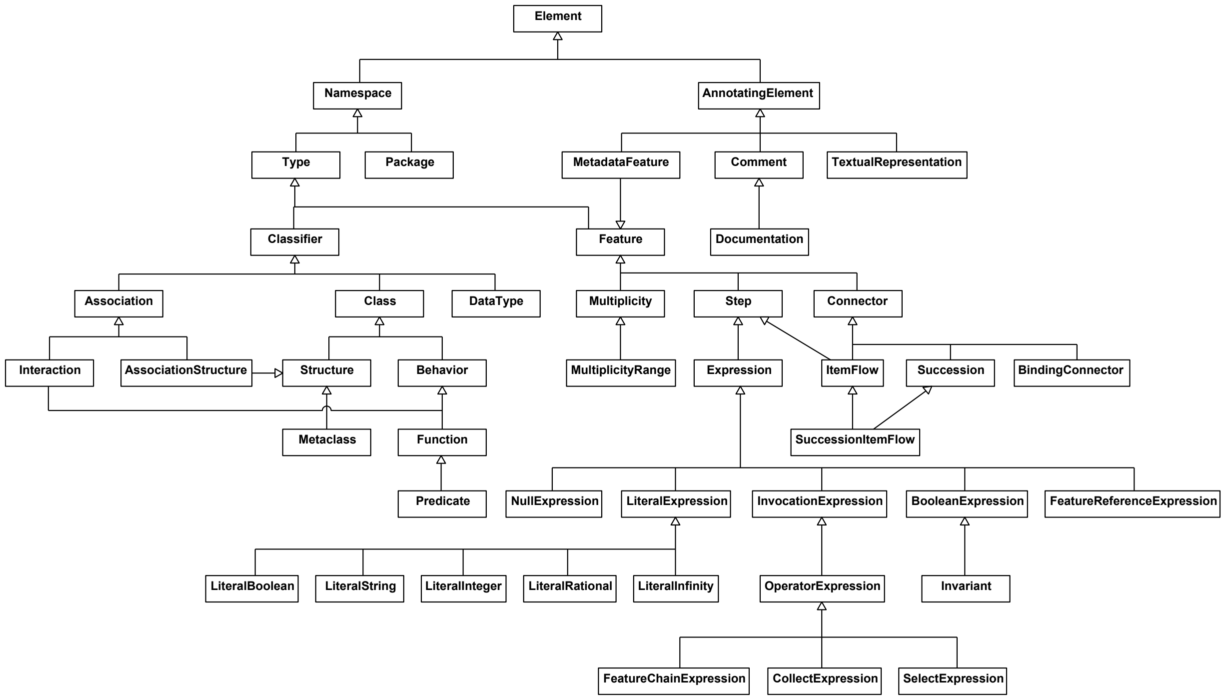

Figure 3. KerML Element Hierarchy

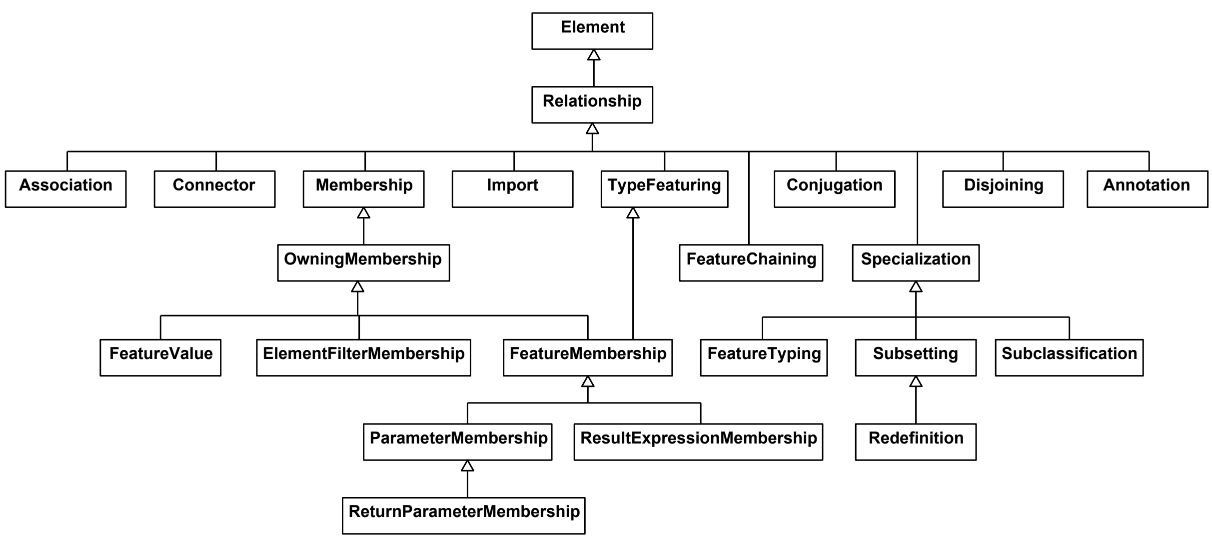

Figure 4. KerML Relationship Hierarchy

7.1.5 Semantics

KerML semantics is specified by a combination of mathematics and model libraries, as illustrated in Fig. 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.

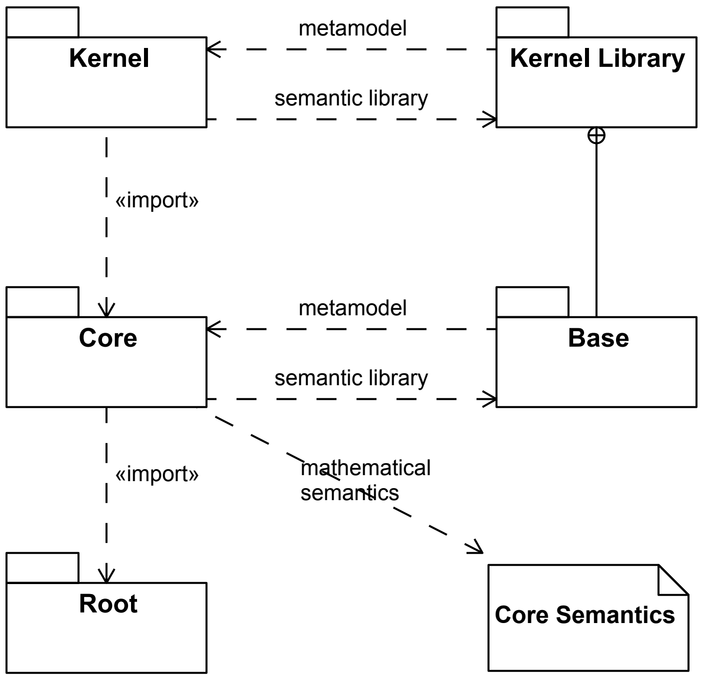

Figure 5. KerML Semantic Layers

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. Semantics are 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 Namespaces. Namespaces can assign unique names to Namespace members, but support multiple aliases per Element. They also support Import of Elements from other Namespaces, 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 `elementId` 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 `elementId` 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 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 `elementId` value shall be considered versions of the same constituent 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.

Every Element may have a `name` and or a `shortName`. While this specification makes no formal distinction between then, the intent is that the `name` of an Element should be fully descriptive, particularly in the context of the definition of the Element, while the `shortName`, if given, should be an abbreviated name useful for referring to the Element. For further discussion of naming, see 7.2.4.

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 known 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, see 7.4.4 and 7.4.5).

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 rooted in 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) =
    ( '<' e.shortName = NAME '>' )? ( e.name = NAME )?

Element : Element =
    'element' Identification(this) ElementBody(this)

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

OwnedElement (e : Element) =
      e.ownedRelationship += OwnedRelationship(e)
    | e.ownedRelationship += OwnedCommentAnnotation
    | e.ownedRelationship += OwnedTextualRepresentationAnnotation
    | e.ownedRelationship += OwnedMetadataFeatureAnnotation
```

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 `shortName` and/or `name` for it, in that order. Both the `shortName` and the `name` are have the same lexical structure (see 7.1.2.4), but the `shortName` is distinguished by being surrounded by the delimiting characters < and >. Note that the notation does not have any provision for specifying the `identifier` or other `aliasIds` of an Element, since these are expected to be managed by the underlying modeling tooling.

```
element <e145> MyName;
```

Note that it is not required to specify either a `shortName` 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 Comments (see 7.2.3.2.1), using the keyword **`comment`** or the keyword **`doc`**. The containing Element becomes the `owningRelatedElement` for the Annotation Relationship to the Comment or Documentation.

- Owned TextualRepresentations (see 7.2.3.2.1), using the keyword **`rep`** or **`language`**. The containing Element becomes the `ownedRelatedElement` for the Annotation Relationship to the TextualRepresentation.
- Owned MetadataFeature (see 7.4.12), using the keyword **`metadata`** or the symbol `@`. The containing Element becomes the `ownedRelatedElement` for the Annotation Relationship to the MetadataFeature.

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

7.2.2.2.2 Relationships

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

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

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

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

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

RelationshipTargetList (r : Relationship) =
    RelationshipTarget(r) ( ',' RelationshipTarget(r) )*

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

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

RelationshipOwnedElement (r : Relationship) =
      r.ownedRelatedElement += OwnedRelatedElement
    | r.ownedRelatedElement += OwnedRelatedRelationship
    | r.ownedRelationship += OwnedCommentAnnotation
    | r.ownedRelationship += OwnedTextualRepresentationAnnotation
    | r.ownedRelationship += OwnedMetadataFeatureAnnotation

OwnedRelatedElement : Element =
    'element' Identification(this) ElementBody(this)

OwnedRelatedRelationship : Relationship =
    'relationship' Identification(this)
    RelationshipRelatedElements(this)
    RelationshipBody(this)
```

A Relationship can be declared using the keyword **`relationship`**. As for a generic Element (see Elements above), a `shortName` and/or a `name` 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 <'1'> A;
element <'2'> B;
```

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

The top-level Elements of a model are implicitly declared within a *root* Namespace for their `shortNames` and `names` (see 7.2.4). If the model is further organized into a complete Namespace structure, then Elements may be identified using qualified names according that structure (see 7.2.4.2.4 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 **`relationship`**. Such Relationships become `ownedRelatedElements` of the containing Relationship (which are always target Elements).
- Owned Comments (see 7.2.3.2.1), using the keyword **`comment`** or the keyword **doc**, as for a generic Element (see 7.2.2.2.1).
- Owned TextualRepresentations (see 7.2.3.2.2), using the keyword **rep** or **language**, as for a generic Element (see 7.2.2.2.1).
- Owned MetadataFeature (see 7.4.12), using the keyword **`metadata`** or the symbol `@`., as for a generic Element (see 7.2.2.2.1).

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 Namespace 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 7.2.2.2.1 above).

```
element A;
element B {
    relationship x {
        element y; // Owned related Element
        relationship from A 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

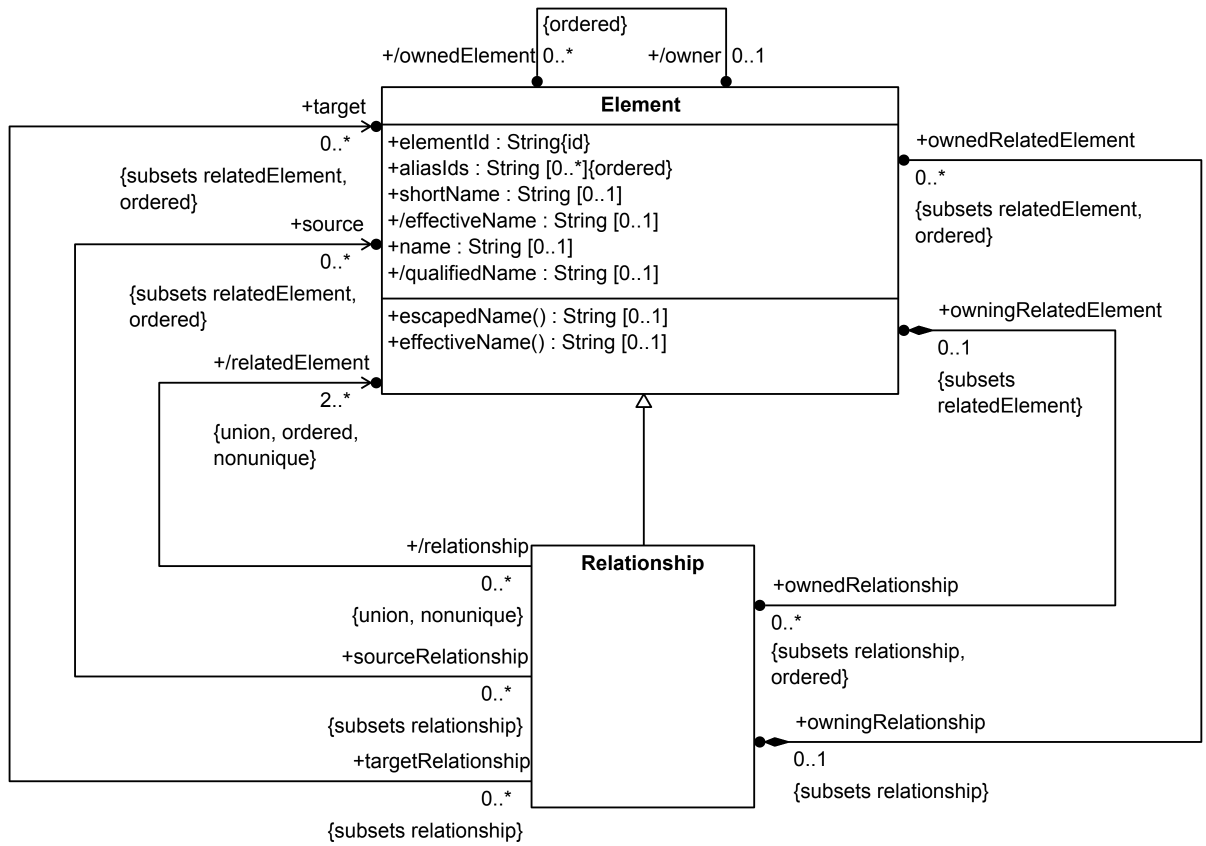

Figure 6. Elements

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

None.

Attributes

aliasIds : String [0..*] {ordered}

Various alternative identifiers for this Element. Generally, these will be set by tools.

/documentation : Documentation [0..*] {subsets ownedElement, annotatingElement, ordered}

The Documentation owned by this Element.

/effectiveName : String [0..1]

The effective name to be used for this Element during name resolution within its `owningNamespace`.

elementId : 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.

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

The `ownedRelationships` of this Element that are Annotations, for which this Element is the `annotatedElement`.

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

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`.

/owner : Element [0..1]

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

owningMembership : OwningMembership [0..1] {subsets owningRelationship, membership}

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

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

The Namespace that owns this Element, derived as the `membershipOwningNamespace` 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.

/qualifiedName : String [0..1]

The full ownership-qualified name of this Element, represented in a form that is valid according to the KerML textual concrete syntax for qualified names (including use of unrestricted name notation and escaped characters, as necessary). The `qualifiedName` is null if this Element has no `owningNamespace` or if there is not a complete ownership chain of named Namespaces from a root Namespace to this Element.

shortName : String [0..1]

An optional alternative name for the Element that is intended to be shorter or in some way more succinct than its primary `name`. It may act as a modeler-specified identifier for the Element, though it is then the responsibility of the modeler to maintain the uniqueness of this identifier within a model or relative to some other context.

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

The `textualRepresentations` that annotate this Element.

Operations

effectiveName() : String [0..1]

Return the effective name for this Element. By default this is the same as its `name`, but, for certain kinds of Elements, this may be overridden if the Element `name` is empty (e.g., for redefining Features).

```
body: name
```

escapedName() : String [0..1]

Return `effectiveName`, if that is not null, otherwise `shortName`, if that is not null, otherwise null. If the returned name is non-null, it is returned as-is if it has the form of a basic name, or, otherwise, represented as a restricted name according to the lexical structure of the KerML textual notation (i.e., surrounded by single quote characters and with special characters escaped).

Constraints

elementDocumentation

The `documentation` of an Element are its `ownedElements` that are Documentation.

```
documentation = ownedElement->selectByKind(Documentation)
```

elementOwnedElements

The `ownedElements` of an Element are the `ownedRelatedElements` of its `ownedRelationships`.

```
ownedElement = ownedRelationship.ownedRelatedElement
```

elementQualifiedName

If this Element does not have an `owningNamespace`, then its `qualifiedName` is empty. If the `owningNamespace` of this Element is a root Namespace, then the `qualifiedName` of the Element is the escaped name of the Element (if any). If the `owningNamespace` is non-empty but not a root Namespace, then the `qualifiedName` of this Element is constructed from the `qualifiedName` of the `owningNamespace` and the escaped name of the Element, unless the `qualifiedName` of the `owningNamespace` is empty, in which case the `qualifiedName` of this Element is also empty.

```
qualifiedName =
    if owningNamespace = null then null
    else if owningNamespace.owner = null then escapedName()
    else if owningNamespace.qualifiedName = null then null
    else owningNamespace.qualifiedName + '::' + escapedName()
    endif endif endif
```

elementOwnedAnnotation

The `ownedAnnotations` of an Element are its `ownedRelationships` that are Annotations.

```
ownedAnnotation = ownedRelationship->selectByKind(Annotation)->
    select(a | a.annotatedElement = self)
```

elementOwner

The `owneder` of an Element is the `owningRelatedElement` of its `owningRelationship`.

```
owner = owningRelationship.owningRelatedElement
```

elementEffectiveName

The `effectiveName` of an Element is given by the result of the `effectiveName()` operation.

```
effectiveName()
```

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

None.

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`.

If an AnnotatingElement is an `ownedMember` of a Namespace (see 7.2.4) and has no `annotations`, then its `owningNamespace` is considered to be its `annotatedElement` without the need for an Annotation relationship.

Specific kinds of AnnotatingElements include Comments and TextualRepresentations. A further kind of AnnotatingElement, a MetadataFeature for user-defined metadata, is defined in the Kernel layer (see 7.4.12).

Comments and Documentation

A Comment is an AnnotatingElement with a textual `body` that in some way describes its `annotatedElement`. Documentation is a specialization of Comment that has the special status of providing *documentation* for the `documentedElement`. A Documentation Comment is always an ownedElement of its `documentedElement`.

Textual Representation

A TextualRepresentation is an AnnotatingElement whose textual `body` represents the `representedElement` in a given `language`. Similarly to Documentation, a TextualRepredentation must be an `ownedElement` of its `representedElement`.

If the named `language` of a TextualRepresentation 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 5 (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 5, 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 Annotation is 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 5. 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 : Comment =
    'comment' Identification(this)
    ( 'about' annotation += Annotation
      { ownedRelationship += annotation }
      ( ',' annotation += Annotation
        { ownedRelationship += annotation } )*
    )?
    body = REGULAR_COMMENT

Annotation : Annotation =
    annotatedElement = [QualifiedName]

PrefixComment : Comment =
    ( 'comment' Identification(this)? )?
    body = PREFIX_COMMENT

Documentation : Comment =
    'doc' identification(this)
    body = REGULAR_COMMENT

OwnedCommentAnnotation : Annotation =
      ownedRelatedElement += OwnedComment(this)
    | ownedRelatedElement += OwnedDocumention(this)

OwnedComment (a : Annotation) : Comment =
    ( 'comment' Identification(this) )?
    body = REGULAR_COMMENT
    { annotation += a }

OwnedDocumentation (a : Annotation) : Documentation =
    'doc' Identification(this)
    body = REGULAR_COMMENT
    { annotation += a }
```

The full declaration of a Comment begins with the keyword **`comment`**, optionally followed by a `shortName` and/or `name` (see 7.2.2.2.1). 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 7.1.2.3).

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

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

```
namespace N {
    comment C /* This is a comment about N. */

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

If the Comment is an `ownedMember` of a Namespace (see 7.2.4), then a special notation can be used in which the Comment text is placed immediately *before* the notation of the `annotatedElement`. Such prefix Comment text is surrounded by `/**` and `*/` delimiters, but a prefix Comment is otherwise declared the same as a regular Comment, except without any **`about`** part. A prefix Comment shall never be the last `ownedMember` of its Namespace, so that it always has a *lexically next* Relationship, defined as the `ownedRelationship` of the Namespace that is immediately after the `owningMembership` of the Comment. An implicit Annotation is added as an `ownedRelationship` of the Comment, with the `annotatingElement` being the Comment and the `annotatedElement` determined as follows:

- If the lexically next Relationship is an OwningMembership, then the `ownedMemberElement` of that OwningMembership.
- Otherwise, the lexically next Relationship itself.

```
namespace P {
    comment pre /** This is a prefix comment about Q. */
    namespace Q;

    /** This is a prefix comment about the following Import. */
    import Q::*;
}
```

A Documentation Comment is notated similarly to a regular Comment, but using the keyword **`doc`** rather than **`comment`**. The `documentingElement` of a Documentation is always the owning Element of the Documentation.

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

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

1. Remove the initial "`/*`" and final "`*/`" characters (or initial "`/**`" and final "`*/`" characters for prefix Comments).

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:

```
namespace 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.

7.2.3.2.2 Textual Representation

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

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

TextualRepresentation : TextualRepresentation =
    ( 'rep' Identification(this )?
    'language' language = STRING_VALUE body = REGULAR_COMMENT
```

A TextualRepresentation is notated similarly to a Documentation Comment (see 7.2.3.2.1), but with the keyword **`rep`** used instead of **`comment`**. As for Documentation, a TextualRepresentation is always owned by its `representedElement`. In particular, if the TextualRepresentation is an `ownedMember` of a Namespace (see 7.2.4), the `representedElement` shall be the containing Namespace. A TextualRepresentation declaration must also specify the `language` as a literal string following the keyword **`language`**. If the TextualRepresentation has no `shortName` or `name`, then the **`rep`** keyword can also be omitted.

```
class C {
    feature x: Real;
    inv x_constraint {
        rep inOCL 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.

Release Note. The final submission may include a means to allow nested comments.

7.2.3.3 Abstract Syntax

7.2.3.3.1 Overview

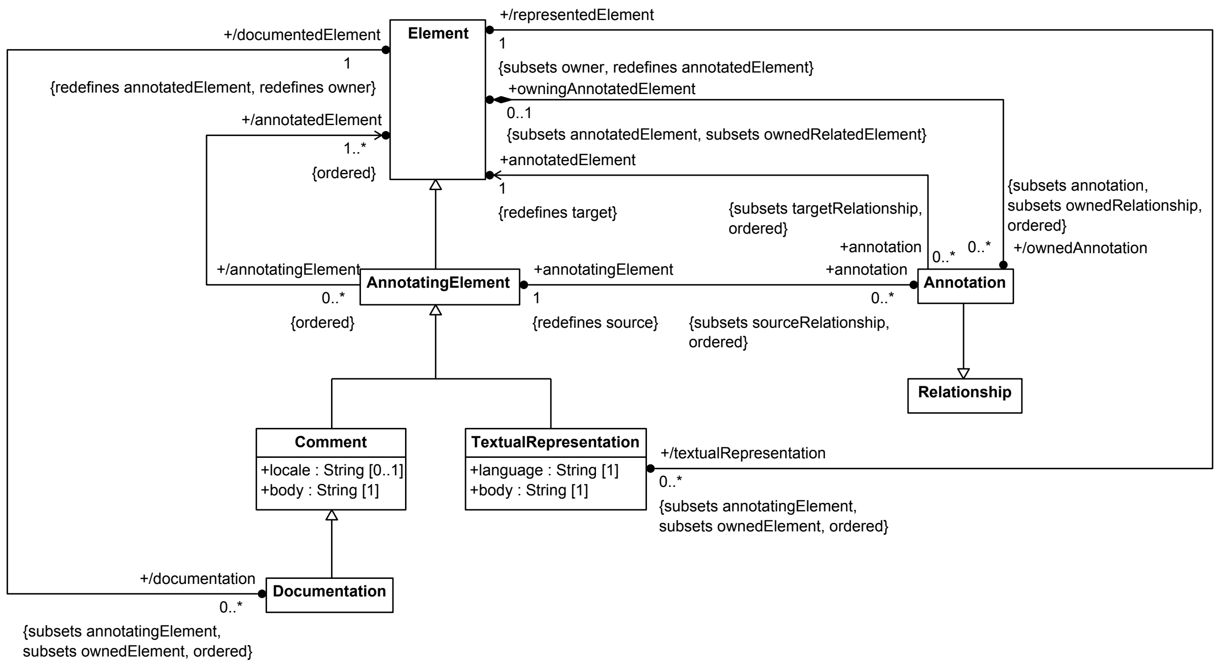

Figure 7. Annotation

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 [1..*] {ordered}

The Elements that are annotated by this AnnotatingElement. If `annotation is not empty, this is derived as the annotatedElements of the annotations. If annotation, then it is derived as the owningNamespace of the AnnotatingElement.`

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

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

Operations

No operations.

Constraints

annotatingElementAnnotatedElement

[no documentation]

```
annotatedElement =
 if annotation->notEmpty() then annotation.annotatedElement
 else owningNamespace endif
```

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, ownedRelatedElement}

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

Operations

No operations.

Constraints

None.

7.2.3.3.4 Comment

Description

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

General Classes

AnnotatingElement

Attributes

body : String

The annotation text for the Comment.

locale : String [0..1]

Identification of the language of the `body` text and, optionally, the region and/or encoding. The format shall be a POSIX locale conformant to ISO/IEC 15897, with the format
`[language[_territory][.codeset][@modifier]]`.

Operations

No operations.

Constraints

None.

7.2.3.3.5 Documentation

Description

Documentation is a Comment that specifically documents a `documentedElement`, which must be its `owner`.

General Classes

Comment

Attributes

/documentedElement : Element {redefines owner, annotatedElement}

The Element that is documented by this Documentation.

Operations

No operations.

Constraints

None.

7.2.3.3.6 TextualRepresentation

Description

A TextualRepresentation is an AnnotatingElement whose `body` represents the `representedElement` in a given `language`. The `representedElement` must be the `owner` of the TextualRepresentation. 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

The annotation text for the Comment.

language : String

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

/representedElement : Element {subsets owner, redefines annotatedElement}

The Element that is represented by this TextualRepresentation.

Operations

No operations.

Constraints

None.

7.2.4 Namespaces

7.2.4.1 Namespaces Overview

Memberships

A Namespace is an Element that contains other Elements via Membership Relationships with those Elements. The Namespace that is the `source` of a Membership Relationship shall also be its `owningRelatedElement`, known as the `membershipOwningNamespace` of the Membership. The Memberships for which a Namespace is the `membershipOwningNamespace` are the `ownedMemberships` of the Namespace.

The `target` of a Membership can be any kind of Element, known as the `memberElement` of the Membership. If the Membership is an OwningMembership, then the `memberElement` property is redefined as the composite `ownedMemberElement` property, which shall also be the only `ownedRelatedElement` of the `OwningMembership`.

A Namespace may also have Import Relationships to other Namespaces. The Namespace that is the `source` of an Import Relationship shall also be its `owningRelatedElement`, known as the `importOwningNamespace` of the Import. The Namespace that is the `target` of an Import Relationship is known as the `importedNamespace` of the Import. The `importOwningNamespace` of an Import shall be different than its `importedNamespace`.

The visible Memberships of the `importedNamespace` of an Import shall become `importedMemberships` of the `importOwningNamespace`. The *visible* Memberships of a Namespace shall comprise at least the following:

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

The complete set of `memberships` of a Namespace include all its `ownedMemberships` and all its `importedMemberships`. Subclasses of Namespace may define additional Memberships to be included in the `memberships` of that kind of Namespace (for instance, the `memberships` of a Type also include its `inheritedMemberships`—see 7.3.2) and which of those are visible (e.g., public `inheritedMemberships`).

The members of a Namespace are the memberElements of all the memberships of the Namespace. The ownedMembers are the ownedMemberElements of all the ownedMemberships of the Namespace.

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

Naming

Each member of a Namespace can optionally be given one or more names *relative to* that Namespace. The *names* of a member of a Namespace shall consist of the memberNames and memberShortNames specified for all the Memberships by which the member is related to the Namespace. Note that the same Element may be related to a Namespace by multiple Memberships, allowing the Element to have multiple, different names relative to that Namespace.

For an OwningMembership, the ownedMemberName and ownedMemberShortName are derived as the name and shortName of the ownedMemberElement, respectively. For Memberships that are not OwningMemberships, the memberName and memberShortName may be set independently of the name and shortName of the memberElement, and therefore provide *aliases* for the memberElement relative to the membershipOwningNamespace.

The names of all the ownedMembers of a Namespace shall be distinct from each other. Further, if a name (i.e., either the memberName or the memberShortName) of any visible Membership of an importedNamespace conflicts with the name of any of any ownedMember of the importOwningNamespace, or with the name of any visible Membership of the importedNamespace of any other Import, then that Membership shall be considered *hidden,* and it shall *not* be included in the set of importedMemberships of the importOwningNamespace.

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

Implementation Note. The pilot implementation does not current check to see if one importedMembership is hidden by another importedMembership. Instead, if there are two importedMemberships with the same name, and they are not hidden by an ownedMembership (or inheritedMembership for a type), name resolution will find first of the importMemberships with that name.

Release Note. The current rules for Membership distinguishibility in a Namespace require that all names be distinct from each other. This may be loosened in the final submission to allow overloading of behavioral Elements with the same name when these can be distinguished by having different parameter signatures.

If an Element is a member of a Namespace, then any name for that Element relative to the Namespace is known as an *unqualified name* for that Element in the Namespace. If the containing Namespace is not a root Namespaec, then the *qualified name* for the member Element consists of a name for the containing Namespace, known as the *qualifier,* followed by an unqualified name for the Element. Since a Namespace is an Element that may itself be a member of another Namespace, a qualifier may be a qualified name. Therefore, a qualified name of an Element, in gneral, has the form of a list of unqualified names of Namespaces, each relative to the previous one, followed by the unqualified name of the Element in the final Namespace.

Since Namespaces 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 Namespace does not have any name, then its members will have no qualified names, even if they are themselves named.

Regardless of whether a root Namespace is named, the name of a top-level Element is *not* qualified by the name of its containing root Namespace. This is because the name resolution rules consider all top-level Elements to be directly visible in the global scope without qualification (see 7.2.4.2.4). Therefore, the *fully qualified* name of an Element relative to a root Namespace always begins with the name of a top-level Element in the root Namespace, without regard to the name (if any) of the root Namespace.

7.2.4.2 Concrete Syntax

7.2.4.2.1 Namespaces

```
RootNamespace : Namespace =
    NamespaceBodyElement(this)*

Namespace : Namespace =
    NamespaceDeclaration(this) NamespaceBody(this)

NamespaceDeclaration (n : Namespace) : Namespace =
    'namespace' Identification(n)

NamespaceBody (n : Namespace) =
    ';' | '{' NamespaceBodyElement(n)* '}'
```

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

The declaration of a root Namespace is implicit and no identification of it is provided in the KerML textual notation. Instead, the body of a root Namespace is given simply by the list of representations of its top-level elements.

```
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 Namespace that is not a root Namespace, and does not represent any more specialized modeling construct (such as a Type—see 7.3.2) is declared using the keyword **`namespace`**, optionally followed by a `shortName` and/or `name` (see 7.2.2.2.1). The body of the Namespace is notated as a list of representations of the content of the Namespace delimited between curly braces `{...}`. If the Namespace is empty, then the body may be omitted and the declaration ended instead with a semicolon.

```
namespace <'1.1'> N1; // This is an empty namespace.
namespace <'1.2'> N2 {
    doc /* This is an example of a namespace body. */
    class C;
    datatype D;
    feature f : C;
    namespace N3; // This is a nested namespace.
}
```

7.2.4.2.2 Namespace Bodies

```
NamespaceBodyElement (n : Namespace) =
      n.ownedRelationship += NamespaceMember
    | n.ownedRelationship += AliasMember
    | n.ownedRelationship += Import

MemberPrefix (m : Membership) =
    ( m.visibility = VisibilityIndicator )?

NamespaceMember : OwningMembership =
      NonFeatureMember
    | NamespaceFeatureMember

NonFeatureMember : OwningMembership =
    MemberPrefix(this)
    ownedMemberElement = NonFeatureElement(this)

NamespaceFeatureMember : Membership =
    MemberPrefix(this)
    ownedMemberElement = FeatureElement(this)

AliasMember : Membership =
    MemberPrefix(this)
    'alias' ( '<' memberShortName = Name '>' )?
    ( memberName = Name )?
    'for' memberElement = [QualifiedName] ';'

Import : Import =
    ( visibility = VisibilityIndicator )?
    'import' ( isImportAll ?= 'all' )?
        ( ImportedNamespace(this)
        | ImportedFilterPackage(this) ) ';'

ImportedNamespace (i : Import) =
    ( i.importedNamespace = [QualifiedName] '::' )?
    ( i.importedName = Name | '*' )
    ( '::' i.isRecursive ?= '**' )?

ImportedFilterPackage (i :Import) =
    i.ownedRelatedElement += FilterPackage

FilterPackage : Package =
    ownedRelationship += FilterPackageImport
    ( ownedRelationship += FilterPackageMember )+

FilterPackageImport : Import =
    ImportedNamespace (this)

FilterPackageMember : ElementFilterMembership =
    '[' condition = OwnedExpression ']'
    { visibility = 'private' }

VisibilityIndicator : VisibilityKind =
    'public' | 'private' | 'protected'
```

Declaring an Element within the body of a Namespace denotes that the Element is an `ownedMember` of the Namespace—that is, that there is an `ownedMembership` of the Namespace that is an OwningMember with the Element as its `ownedMemberElement`. The `name` and `shortName` given for the Element (if any) become the `ownedMemberName` and `ownedMemberShortName` of the OwningMembership, respectively. The `visibility` of the Membership can also be specified by placing one of the keywords **`public`**, **`protected`** or **`private`** before the Element declaration. If no visibility is specified, the default is **`public`**. For Namespaces other than Types, **`protected`** visibility is equivalent to **`private`**. For Types, **`protected`** visibility has a special meaning relating to `member` inheritance (see 7.3.2).

```
namespace N {
    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 the alias `memberShortName` and/or `memberName`, with a qualified name (see 7.2.4.2.4) identifying the Element given after the keyword **`for`**. This denotes an `ownedMembership` of the containing Namespace that is *not* an OwningMembership, with the identified Element as an unowned `memberElement`. The `visibility` of the Membership can be specified as for an `ownedMember`.

```
namespace N1 {
    class A;
    class B;
    alias <C> CCC for B;
}
```

An `ownedImport` of a Namespace is denoted using the keyword **`import`** followed by a qualified name (see 7.2.4.2.4). This specifies an Import whose `importedNamespace` is the qualification part of the qualified name and whose `importedMemberName` is given by the the unqualified name. If the name given for the **`import`** is unqualified, then the `importedNamespace` shall be null and the given name shall be resolved in the scope of the Namespace owning the Import (see 7.2.4.2.4).

Such an Import results in the Membership of the `importedNamespace` whose `memberName` or `memberShortName` is the given `importedMemberName` becoming an `importedMembership` of the Namespace owning the Import. That is, the `memberElement` of this Membership becomes an imported `member` of the importing Namespace. Note that the `importedMemberName` may be an alias of the imported Element in the `importedNamespace`, in which case the Element is still imported with that name.

```
namespace N2 {
    import N1::A;
    import N1::C; // Imported with name "C".
    namespace M {
        import C; // "C" is re-imported from N2 into M.
    }
}
```

If the qualified name in an import is follow followed by `"::*"`, then the entire qualified name shall identify the `importedNamespace` and the `importedMemberName` shall be null. In this case, all visible Memberships of the `importedNamespace` of the Import shall become `importedMemberships` of the importing Namespace.

```
namespace N3 {
    // Memberships A, B and C are all imported from N1.
    import N1::*;
}
```

If the qualified name of an **`import`**, with or without a "`::*`", is further followed by "`::**`", then the import shall be *recursive.* Such an import is equivalent to importing all Memberships as described above, followed by further recursively importing from each imported member that is itself a Namespace.

```
namespace N4 {
    class A;
    class B;
    namespace M {
        class C;
    }
}
namespace N5 {
    import N4::**;
    // The above recursive import is equivalent to all
    // of the following taken together:
    //     import N4;
    //     import N4::*;
    //     import N4::M::*;
}
namespace N6 {
    import N4::*::**;
    // The above recursive import is equivalent to all
    // of the following taken together:
    //     import N4::*;
    //     import N4::M::*;
    // (Note that N4 itself is not imported.)
}
```

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`**.

```
namespace N7  {
    // The imported membership is visible outside N7.
    public import N1::A;

    // None of the imported memberships are visible outside of N7.
    private import N4::*;
}
```

An Import may also be declared with one or more `filterConditions`, given as model-level evaluable Boolean Expressions (see 7.4.8), listed after the `importedNamespace` specification, each surrounded by square brackets `[... ]`. Such a filtered Import is equivalent to importing an implicit Package that then both imports the given `importedNamespace` and has all the given `filterConditions`. The effect is such that, for a filtered Import, Memberships shall be imported from the `importedNamespace` if and only if they satisfy all the given `filterConditions`. (While filtered Imports may be used in any Namespace, Packages and `filterConditions` are actually Kernel-layer concepts, because Expressions are only defined in that layer. See 7.4.13.)

```
namespace N8 {
    import Annotations::*;

    // Only import elements of NA that are annotated as Approved.
    import NA::*[@Approved];
}
```

A Comment (see 7.2.3.2.1), including Documentation, declared within a Namespace body also becomes an `ownedMember` of the Namespace. If no `annotatedElements` are specified for the Comment, then, by default, the Comment is considered to be about the containing Namespace.

```
namespace N9 {
    class A;
    comment Comment1 about A
        /* This is a comment about class A. */

    comment Comment 2
        /* This is a comment about namespace N9. */

    /* This is also a comment about namespace N9. */

    doc N9_Doc
        /* This is documentation about namespace N9. */
}
```

A prefix Comment is always an `ownedMember` of a Namespace, and it is about the body Element lexically after the Comment, as described in 7.2.3.2.1.

```
namespace N10 {
    /** This is acomment about the following comment on member B. */
    /** This is a comment about member B. */
    private class B;

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

    /** This is a comment about the import of N4. */
    import N4::*;
}
```

7.2.4.2.3 Namespace Elements

```
NonFeatureElement : Element =
      Element
    | Relationship
    | Comment
    | PrefixComment
    | Documentation
    | TextualRepresentation
    | MetadataFeature
    | Namespace
    | Type
    | Classifier
    | DataType
    | Class
    | Structure
    | Metaclass
    | Association
    | AssociationStructure
    | Interaction
    | Behavior
    | Function
    | Predicate
    | Multiplicity
    | Package
    | Specialization
    | Conjugation
    | Subclassification
    | Disjoining
    | FeatureTyping
    | Subsetting
    | Redefinition
    | TypeFeaturing

FeatureElement : Feature =
      Feature
    | Step
    | Expression
    | BooleanExpression
    | Invariant
    | Connector
    | BindingConnector
    | Succession
    | ItemFlow
    | SuccessionItemFlow
```

A Namespace 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 Multiplicity, Package and the various specialized kinds of Classifiers defined in the Kernel (see 7.4). This division is convenient because, in the Core, Feature Elements may be related to Types using a specialized FeatureMembership Relationship, while non-Feature Elements are always related to Types using the same generic Membership Relationship used with non-Type Namespaces.

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

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

1. For each Element that is directly contained in a Namespace, but is *not* a member of the Namespace, the `shortName` and `effectiveName` of the Element, if non-null, locally resolve to that Element.
2. For each `membership` of a Namespace, the `memberShortName` and `memberName` of the Membership, if it non-null, locally resolve to the `memberElement` of the Membership.

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

The *visible resolution* of a name restricts the `memberships` in the second step to those that are visible outside the Namespace. Note that resolution of names in the first step is not restricted by visibility.

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

1. If the name locally resolves to an Element directly contained in the Namespace, then it fully resolves to that Element.
2. If there is no such Element, then:
    - If the Namespace is *not* a root Namespace, then the name resolution continues with the `owningNamespace` of the Namespace.
    - If the Namespace *is* a root Namespace, 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 a `shortMemberName` or `memberName` equal to the simple name, then the name resolves to the `memberElement` of that Membership.
2. 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 resolve a given simple name. In this case, one of these Memberships is chosen for the resolution of the name, but which one is chosen is not otherwise determined by this specification.

Implementation Note. The pilot implementation currently only resolves the `ownedMemberNames` of OwningMemberships in the global namespace. Short names and aliases are *not* resolved.

A qualified name is always used to identify an Element that is a `target` Element of some Relationship. The *context* Namespace is the nearest Namespace 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 Namespace.
- If the context Relationship *is* a Membership or an Import, then
    - If the context Namespace is *not* a root Namespace, then the local namespace is the `owningNamespace` of the context Namespace.
    - If the context Namespace *is* a root Namespace, then the local namespace is the global namespace for the context Namespace.

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 Namespace that is the visible resolution of the name relative to the Namespace identified by the previous segment. The qualified name then resolves to the resolution of its last segment name.

Note. In the concrete syntax productions found in various other subclauses, the notation `[QualifiedName]` is used to signify that a `QualifedName` shall be parsed, then that name shall be resolved into a reference to an Element, per the rules given in this subclause, and that reference shall be inserted into the abstract syntax as specified in the production, not the `QualifiedName` itself (see also 7.1.3, Table 3).

7.2.4.3 Abstract Syntax

7.2.4.3.1 Overview

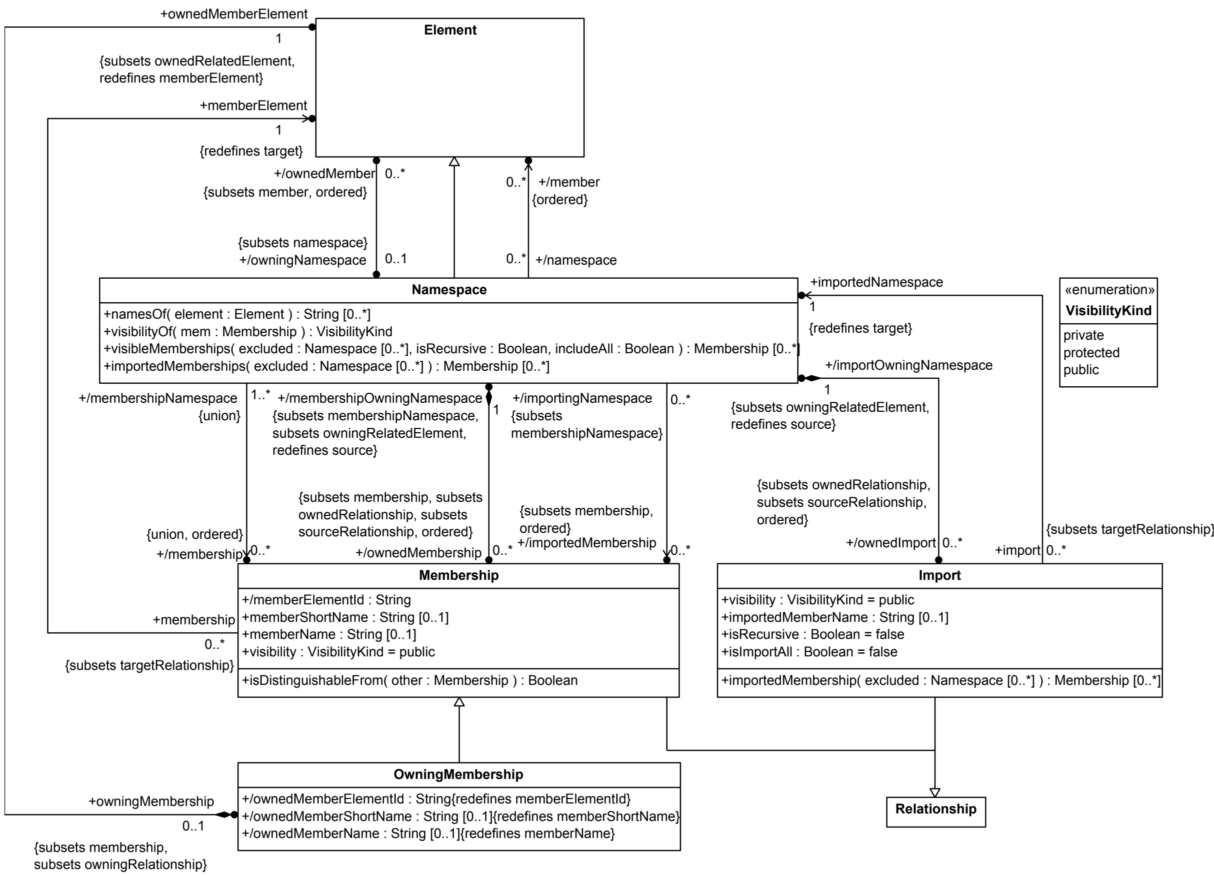

Figure 8. Namespaces

7.2.4.3.2 Import

Description

An Import is a Relationship between an `importOwningNamespace` in which one or more of the visible Memberships of the `importedNamespace` become `importedMemberships` of the `importOwningNamespace`. If `isImportAll = false` (the default), then only public Memberships are considered "visible". If `isImportAll = true`, then all Memberships are considered "visible", regardless of their declared `visibility`.

If no `importedMemberName` is given, then all visible Memberships are imported from the `importedNamespace`. If `isRecursive = true`, then visible Memberships are also recursively imported from all visible `ownedMembers` of the Namespace that are also Namespaces.

If an `importedMemberName` is given, then the Membership whose `effectiveMemberName` is that name is imported from the `importedNamespace`, if it is visible. If `isRecursive = true` and the imported `memberElement` is a Namespace, then visible Memberships are also recursively imported from that Namespace and its owned sub-Namespaces.

General Classes

Relationship

Attributes

importedMemberName : String [0..1]

The `effectiveMemberName` of the Membership of the `importedNamspace` to be imported. If not given, all public Memberships of the `importedNamespace` are imported.

importedNamespace : Namespace {redefines target}

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

/importOwningNamespace : Namespace {subsets owningRelatedElement, redefines source}

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

isImportAll : Boolean

Whether to import memberships without regard to declared visibility.

isRecursive : Boolean

Whether to recursively import Memberships from visible, owned sub-namespaces.

visibility : VisibilityKind

The visibility level of the imported `members` from this Import relative to the `importOwningNamespace`.

Operations

importedMembership(excluded : Namespace [0..*]) : Membership [0..*]

Returns the Memberships of the `importedNamespace` whose `memberElements` are to become imported `members` of the `importOwningNamespace`. By default, this is the set of publicly visible Memberships of the `importedNamespace`, but this may be overridden in specializations of Import. (The `excluded` parameter is used to handle the possibility of circular Import Relationships.)

```
body: let exclusions : Set(Namespace) =
    excluded->including(importOwningNamspace) in
let visibleMemberships : Sequence(Membership) =
    importedNamespace.visibleMemberships(exclusions, false, isImportAll) in
let memberships : Sequence(Membership) =
    if importedMemberName = null then visibleMemberships
    else visibleMemberships->select(effectiveMemberName = importedMemberName)
    endif in
if not isRecursive then memberships
else memberships->union(
        memberships.ownedMember->selectAsKind(Namespace).
        visibleMemberships(exclusions, true, isImportAll))
endif
```

Constraints

None.

7.2.4.3.3 Membership

Description

Membership is a Relationship between a Namespace and an Element that indicates the Element is a `member` of (i.e., is contained in) the Namespace. Any `memberNames` specify how the `memberElement` is identified in the Namespace and the `visibility` specifies whether or not the `memberElement` is publicly visible from outside the Namespace.

If a Membership is an OwningMembership, then it owns its `memberElement`, which becomes an `ownedMember` of the `membershipOwningNamespace`. Otherwise, the `memberNames` of a Membership are effectively aliases within the `membershipOwningNamespace` for an Element with a separate OwningMembership in the same or a different Namespace.

General Classes

Relationship

Attributes

memberElement : Element {redefines target}

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

/memberElementId : String

The `elementId` of the `memberElement`.

memberName : String [0..1]

The name of the `memberElement` relative to the `membershipOwningNamespace`.

/membershipOwningNamespace : Namespace {subsets membershipNamespace, owningRelatedElement, redefines source}

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

memberShortName : String [0..1]

The short name of the `memberElement` relative to the `membershipOwningNamespace`.

visibility : VisibilityKind

Whether or not the Membership of the `memberElement` in the `membershipOwningNamespace` is publicly visible outside that Namespace.

Operations

isDistinguishableFrom(other : Membership) : Boolean

Whether this Membership is distinguishable from a given `other` Membership. By default, this is true if this Membership has no `memberShortName` or `memberName`; or each of the `memberShortName` and `memberName` are different than both of those of the `other` Membership; or neither of the metaclasses of the `memberElement` of this

Membership and the `memberElement` of the `other` Membership conform to the other. But this may be overridden in specializations of Membership.

```
body: not (memberElement.oclKindOf(other.memberElement.oclType()) or
     other.memberElement.oclKindOf(memberElement.oclType())) or
(shortMemberName = null or
    (shortMemberName <> other.shortMemberName and
     shortMemberName <> other.memberName)) and
(memberName = null or
    (memberName <> other.shortMemberName and
     memberName <> other.memberName)))
```

Constraints

None.

7.2.4.3.4 Namespace

Description

A Namespace is an Element that contains other Elements, known as its `members`, via Membership Relationships with those Elements. The `members` of a Namespace may be owned by the Namespace, aliased in the Namespace, or imported into the Namespace via Import Relationships with other Namespaces.

A Namespace can provide names for its `members` via the `memberNames` specified by the Memberships in the Namespace. If a Membership specifies a `memberName`, then that is the name of the corresponding `memberElement` relative to the Namespace. Note that the same Element may be the `memberElement` of multiple Memberships in a Namespace (though it may be owned at most once), each of which may define a separate alias for the Element relative to the Namespace.

General Classes

Element

Attributes

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

The Memberships in this Namespace that result from Import Relationships between the Namespace and other Namespaces.

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

The set of all member Elements of this Namespace, derived as the `memberElements` of all `memberships` of the Namespace.

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

All Memberships in this Namespace, including (at least) the union of `ownedMemberships` and `importedMemberships`.

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

The `ownedRelationships` of this Namespace that are Imports, for which the Namespace is the `importOwningNamespace`.

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

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

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

The `ownedRelationships` of this Namespace that are Memberships, for which the Namespace is the `membershipOwningNamespace`.

Operations

importedMemberships(excluded : Namespace [0..*]) : Membership [0..*]

Derive the imported Memberships of this Namespace as the `importedMembership` of all `ownedImports`, excluding those Imports whose `importOwningNamespace` is in the `excluded` set, and excluding Memberships that have distinguisibility collisions with each other or with any `ownedMembership`.

```
body: ownedImport->
    excluding(excluded->contains(importOwningNamespace)).
    importedMembership(excluded)
```

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

Return the names of the given `element` as it is known in this Namespace.

```
body: let elementMemberships : Sequence(Membership) =
    memberships->select(memberElement = element)
in
    memberships.memberShortName->
        union(memberships.memberName)->
        asSet()
```

visibilityOf(mem : Membership) : VisibilityKind

Returns this visibility of `mem` relative to this Namespace. If `mem` is an `importedMembership`, this is the `visibility` of its Import. Otherwise it is the `visibility` of the Membership itself.

```
body: if importedMembership->includes(mem) then
    ownedImport->any(importedMembership(Set{})->includes(mem)).visibility
else if memberships->includes(mem) then
    mem.visibility
else
    VisibilityKind::private
endif
```

visibleMemberships(excluded : Namespace [0..*],isRecursive : Boolean,includeAll : Boolean) : Membership [0..*]

If `includeAll = true`, then return all the Memberships of this Namespace. Otherwise, return only the publicly visible Memberships of this Namespace (which includes those `ownedMemberships` that have a `visibility` of `public` and those `importedMemberships` imported with a `visibility` of `public`). If `isRecursive = true`, also recursively include all visible Memberships of any visible owned Namespaces.

```
body: let publicMemberships : Sequence(Membership) =
    ownedMembership->
        select(visibility = VisibilityKind::public)->
```

```
        union(ownedImport->
            select(visibility = VisibilityKind::public).
            importedMembership(excluded)) in
if not isRecursive then publicMemberships
else publicMemberships->union(publicMemberships->
        selectAsKind(Namespace).
        publicMembership(excluded->including(this), true))
endif
```

Constraints

namespaceMembers

The `members` of a Namespace are the `memberElements` of all its `memberships`.

```
member = membership.memberElement
```

namespaceOwnedMember

The `ownedMembers` of a Namespace are the `ownedMemberElements` of all its `ownedMemberships` that are OwningMemberships.

```
ownedMember = ownedMembership->selectByKind(OwningMembership).ownedMemberElement
```

namespaceOwnedMembership

The `ownedMemberships` of a Namespace are all its `ownedRelationships` that are Memberships.

```
ownedMembership = ownedRelationship->selectByKind(Membership)
```

namespaceOwnedImport

The `ownedImports` of a Namespace are all its `ownedRelationships` that are Imports.

```
ownedImport = ownedRelationship->selectByKind(Import)
```

namespaceDistinguishibility

All `memberships` of a Namespace must be distinguishable from each other.

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

namespaceImportedMembership

The `importedMemberships` of a Namespace are derived using the `importedMemberships()` operation,with no initially `excluded` Namespaces.

```
importedMembership = importedMemberships(Set{})
```

7.2.4.3.5 VisibilityKind

Description

VisibilityKind is an enumeration whose literals specify the visibility of a Membership of an Element in a Namespace outside of that Namespace. Note that "visibility" specifically restricts whether an Element in a Namespace may be referenced by name from outside the Namespace 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

None.

Literal Values

private

Indicates a Membership is not visible outside its owning Namespace.

protected

An intermediate level of visibility between `public` and `private`. By default, it is equivalent to `private` for the purposes of normal access to and import of Elements from a Namespace. However, other Relationships may be specified to include Memberships with `protected` visibility in the list of `memberships` for a Namespace (e.g., Generalization).

public

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

7.2.4.3.6 OwningMembership

Description

An OwningMembership is a Membership that owns its `memberElement` as a `ownedRelatedElement`. The `ownedMemberElementM` becomes an `ownedMember` of the `membershipOwningNamespace`.

General Classes

Membership

Attributes

ownedMemberElement : Element {subsets ownedRelatedElement, redefines memberElement}

The Element that becomes an `ownedMember` of the `membershipOwningNamespace` due to this OwningMembership.

/ownedMemberElementId : String {redefines memberElementId}

The `elementId` of the `ownedMemberElement`.

/ownedMemberName : String [0..1] {redefines memberName}

The `effectiveName` of the `ownedMemberElement`.

/ownedMemberShortName : String [0..1] {redefines memberShortName}

The `shortName` of the `ownedMemberElement`.

Operations

No operations.

Constraints

owningMembershipOwnedMemberShortName

The `ownedMemberName` of an OwningMembership is the `effectiveName` of its `ownedMemberElement`.

```
ownedMemberShortName = ownedMemberElement.shortName
```

owningMembershipOwnedMemberName

The `ownedMemberName` of an OwningMembership is the `effectiveName` of its `ownedMemberElement`.

```
ownedMemberName = ownedMemberElement.effectiveName
```

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 in 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 Specializations between Types (Subclassification for Classifiers, Subsetting and Redefinition for Features). Specialized Types classify all the things their more general Types do (via one or more Specializations). 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 Specialization 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:

- Δ 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 Δ 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*

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

Functions and predicates for sequences are introduced below. Predicates prefixed with `form:` are defined in [fUML], Clause 10 (Base Semantics).

- *length* is a function version of fUML's *sequence-length*.

  $$\forall s, n \;\; n = length(s) \equiv (\texttt{form:sequence-length} \; s \; n)$$

- *at* is a function version of fUML's *in-position-count*.

  $$\forall x, s, n \;\; x = at(s, n) \equiv (\texttt{form:in-position-count} \; s \; n \; x)$$

- *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.

  $$\forall s_1, s_2 \;\; head(s_1, s_2) \Rightarrow \texttt{form:Sequence}(s_1) \land \texttt{form:Sequence}(s_2)$$
  $$\forall s_1, s_2 \;\; head(s_1, s_2) \equiv (length(s_1) \leq length(s_2)) \land (\forall p \in Z^+ \;\; p \geq 1 \land p \leq length(s_1) \Rightarrow at(s_1, p) = at(s_2, p))$$

- *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:

$\forall s_1, s_2\ \ tail(s_1, s_2) \Rightarrow \texttt{form:Sequence}(s_1) \wedge \texttt{form:Sequence}(s_2)$
$\forall s_1, s_2\ \ tail(s_1, s_2) \equiv (length(s1) \leq length(s_2)) \wedge$
$(\forall h, p \in Z^+\ \ (h = length(s_2) - length(s_1)) \wedge (p > h) \wedge (p \leq length(s_2) \Rightarrow at(s_1, p - h) = at(s_2, p))$

- *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:

$\forall s_0, s_1, s_2\ \ concat(s_0, s_1, s_2) \Rightarrow \texttt{form:Sequence}\ (s0) \wedge \texttt{form:Sequence}(s1) \wedge \texttt{form:Sequence}(s2)$

$\forall s_0, s_1, s_2\ \ concat(s_0, s_1, s_2) \equiv (length(s_0) = length(s_1) + length(s_2)) \wedge head(s_1, s_0) \wedge tail(s_2, s_0)$

7.3.2 Types

7.3.2.1 Types Overview

Types and classification

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.

Specialization and other Relationships between Types

Specializations are Relationships between Types, identified as `specific` and `general`, 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` one, referred to syntactically as *inheriting* features from general to specific Types, see below. Specialization Relationships can form cycles, which means all Types in the cycle have the same instances (same extent).Types identify Specializations relating them to more general and specific Types as their `specializations` and `generalizations`, respectively. Specializations to more general Types can be owned by a Type, identified as its `ownedSpecializations`. The *specializations* or *subtypes* (plural, in regular font) of a Type in this specification is short for the Types related to it via `specialization`, and the specializations of those Types, recursively.

Types related by Disjoining do not share instances (instances cannot be in more than one of the extents; the extents are *disjoint*). Types identify Disjoinings relating them as their `disjoiningTypeDisjoinings`. Disjoinings can be owned by a Type, identified as its `ownedDisjoinings`.

Types can be *abstract,* which means that all instances of a Type must also be instances of at least one (possibly indirect) specialized Type (which must not be abstract, that is, must be *concrete*).

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, Features can specialize Classifiers to limit what things the Features can relate to, see FeatureTyping in 7.3.4.1. Classifiers specializing 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. All (chains of) relationships between things in the universe are instances of the Feature *`things`* in the Base model library (see 8.2).

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 Namespaces, 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 `ownedSpecializations`. 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` 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` (for imported Memberships, `protected` visibility is equivalent to `private`). 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

A FeatureMembership is a Relationship between a Type and a Feature that is both a kind of Membership and a kind of TypeFeaturing (see 7.3.4). Features related to a Type via FeatureMembership are identified as the `features` of the Type and are `members` of it. The owning Type is one of the Feature's `featuringTypes` (see 7.3.4). FeatureMemberships are always owned by their Type.

Multiplicity

The number of instances in the extent of a Type (*cardinality*) is constrained by the Type's *multiplicity*. A Multiplicity is a Feature whose values are natural numbers (extended with infinity, see 8.18.1) that are the only ones allowed for the cardinality of its `featuringType` (each 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 it classifies in the modeled universe. For Features that are not end Features (see below), cardinality is the number of values of the Feature for a specific instance of its `featuringTypes`.

Note. See 7.4.11 in Kernel for specifying numeric ranges for multiplicities, rather than each number separately as above.

A Feature with `isEnd = true` is an *end* Feature of its `featuringTypes`. The semantics of Multiplicity is different for end Features. End Features are used primarily in the definition of Associations and Connectors (see 7.4.4 and 7.4.5, respectively, where the semantics of end Features is further discussed).

Conjugation

Conjugation is a Relationship between Types, identified as `originalType` and `conjugatedType`, indicating the `conjugatedType` inherits visible and protected Memberships from the `originalType`, except the direction of input and output Features is reversed. Features 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`. Features with with no direction or direction `inout` in the `originalType` are inherited without change. Types can be `conjugatedTypes` of at most one Conjugation Relationship, and they shall not be the `specific` Type in any Specialization relationship.

7.3.2.2 Concrete Syntax

7.3.2.2.1 Types

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

TypeDeclaration (t : Type) =
    (t.isSufficient ?= 'all' )? Identification(t)
    ( t.ownedRelationship += OwnedMultiplicity )?
    ( SpecializationPart(t) | ConjugationPart(t) )+
    DisjoiningPart(t)?

SpecializationPart (t : Type) =
    SPECIALIZES t.ownedRelationship += OwnedSpecialization
    ( ',' t.ownedRelationship += OwnedSpecialization )*

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

DisjoiningPart (t : Type) =
    'disjoint' 'from' t.ownedRelationship += OwnedDisjoining
    ( ',' t.ownedRelationship += OwnedDisjoining )*

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

TypeBodyElement (t : Type) =
      t.ownedRelationship += NonFeatureMember
    | t.ownedRelationship += FeatureMember
    | t.ownedRelationship += AliasMember
    | t.ownedRelationship += Import
```

Similarly to the generic Namespace 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 shortName and/or name. In addition, a Type declaration defines either one or more ownedSpecializations 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). This may optionally be followed by the definition of one or more ownedDisjoinings (see 7.3.2.3.3).

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 7.4.11 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 Namespace, by listing the members between curly braces {...} (see 7.2.4.2). However, for Types, protected members, indicated using the keyword **protected** instead of **public** or **private**, have special visibility rules, as described in 7.3.2.1. A Feature declared as an ownedMember of a Type is automatically considered to be an ownedFeature of the Type, related by a FeatureMembership, unless its declaration is preceded by the keyword **member**, in which case it is related by regular Membership (see 7.3.2.2.5 for details).

```
type Super specializes Base::Anything {
    private namespace N {
        type Sub specializes Super;
    }
    protected feature f : N::Sub;
    member feature f1 : Super featured by N::Sub;
}
```

7.3.2.2.2 Specialization

```
Specialization : Specialization =
    ( 'specialization' Identification(this) )?
    'subtype' SpecificType(this)
    SPECIALIZES GeneralType(this) ';'

OwnedSpecialization : Specialization =
    GeneralType(this)

SpecificType (s : Specialization) :
      s.specific = [QualifiedName]
    | s.specific += OwnedFeatureChain
      { s.ownedRelatedElement += s.specific }

GeneralType (s : Specialization) :
      s.general = [QualifiedName]
    | s.general += OwnedFeatureChain
      { s.ownedRelatedElement += s.general }
```

A Specialization Relationship is declared using the keyword **`specialization`**, optionally followed by a `shortName` and/or a `name`. The qualified name of the `specific` Type, or a Feature chain (see 7.3.4.2.6) if the `specific` Type is such a Feature, is then given after the keyword **`subtype`**, followed by the qualified name of the `general` Type, or a Feature chain if the `general` Type is such a Feature, after the keyword **`specializes`**. The symbol `:>` can be used interchangeably with the keyword **`specializes`**.

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

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

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

An `ownedSpecialization` of a Type is defined as part of the declaration of the Type, rather than in a separate declaration, by including the qualified name or Feature chain 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 =
    ( 'conjugation' Identification(this) )?
    'conjugate'
    ( conjugatedType = [QualifiedName]
    | conjugatedType = FeatureChain
      { ownedRelatedElement += conjugatedType } )
    CONJUGATES
    ( originalType = [QualifiedName]
    | originalType = FeatureChain
      { ownedRelatedElement += originalType } ) ';'

OwnedConjugation : Conjugation =
      originalType = [QualifiedName]
    | originalType = FeatureChain
      { ownedRelatedElement += originalType }
```

A Conjugation Relationship is declared using the keyword **`conjugation`**, followed by a `shortName` and/or a `name`. The qualified name of the `conjugatedType`, or a Feature chain (see 7.3.4.2.6) if the `conjugatedType` is such a Feature, is then given after the keyword **`conjugate`**, followed by the qualified name of the `originalType`, or a Feature chain (see 7.3.4.2.6) if the `originalType` is such a Feature, 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 conjugate Conjugate1 conjugates Original;
conjugation c2 conjugate Conjugate2 ~ Original;
```

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

```
conjugate Conjugate1 conjugates Original;
conjugate 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 or Feature chain 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 `ownedSpecializaions`.

7.3.2.2.4 Disjoining

```
Disjoining : Disjoining =
    ( 'disjoining' Identification(this) )?
    'disjoint'
    ( typeDisjoined = [QualifiedName]
    | typeDisjoined = FeatureChain
      { ownedRelatedElement += typeDisjoined } )
    'from'
    ( disjoiningType = [QualifiedName]
    | disjoinginType = FeatureChain
      { ownedRelatedElement += disjoiningType } ) ';'

OwnedDisjoining : Disjoining :
      disjoiningType = [QualifiedName]
    | disjoinginType = FeatureChain
      { ownedRelatedElement += disjoiningType }
```

A Disjoining Relationship is declared using the keyword **`disjoining`**, optionally followed by a `shortName` and/or a `name`. The qualified name of the `typeDisjoined`, or a Feature chain (see 7.3.4.2.6) if the `typeDisjoined` is such a Feature, is then given after the keyword **`disjoint`**, followed by the qualified name of the `disjoiningType`, or a Feature chain (see 7.3.4.2.6) if the `disjoiningType` is such a Feature, after the keyword **`from`**.

```
disjoining Disj disjoint A from B;
disjoining disjoint Minor from Adult;
```

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

```
disjoint A from B;
disjoint Minor from Adult;
```

An `ownedDisjoining` of a Type is defined as part of the declaration of the Type, rather than in a separate declaration, by including the qualified name or Feature chain of the `disjoiningType` in a list after the keyword **`disjoint from`**.

```
type C disjoint from A, B;
type Minor disjoint from Adult;
```

7.3.2.2.5 Feature Membership

```
FeatureMember : OwningMembership =
      TypeFeatureMember
    | OwnedFeatureMember

TypeFeatureMember : OwningMembership =
    MemberPrefix 'member' ownedMemberElement = FeatureElement

OwnedFeatureMember : FeatureMembership =
    MemberPrefix ownedMemberFeature = FeatureElement
```

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 Namespaces (see 7.2.4). However, unlike a non-Type Namespace, a Type is the `featuringType` of those of `ownedFeatures`.

A Feature that is declared within the body of a Type is normally an `ownedFeature` of that Type, so it automatically has that type as a `featuringType` (because FeatureMembership is a kind of TypeFeaturing, see 7.3.2). As kinds of Types, this also applies to the bodies of Classifiers (see 7.3.3) and Features (see 7.3.4.2). A Feature may also be aliased in a Type like any other Element (see 7.2.4.2.1), in which case it is related to the aliasing 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 {
    // This declares an ownedFeature using a FeatureMembership.
    feature age[1] : ScalarValues::Integer;

    // This is not a FeatureMembership.
    alias personAlias for person;
}
```

However, if the Feature declaration is preceded by the keyword **`member`**, then the Feature is owned by the containing Type via a Membership Relationship, not a FeatureMembership. In this case, the Feature is *not* an `ownedFeature` of the containing Type, and it does *not* have the containing Type as a `featuringType` and only has the `featuringTypes` declared in its **`featured by`** list, if any (see 7.3.4.2.1 on declaring the `ownedTypings` of a Feature).

```
classifier A;
classifier B {
    // Feature f has B as its featuring type.
    feature f;

    // Feature g has A as its featuring type, not B.
    member feature g featured by A;
}
```

7.3.2.3 Abstract Syntax

7.3.2.3.1 Overview

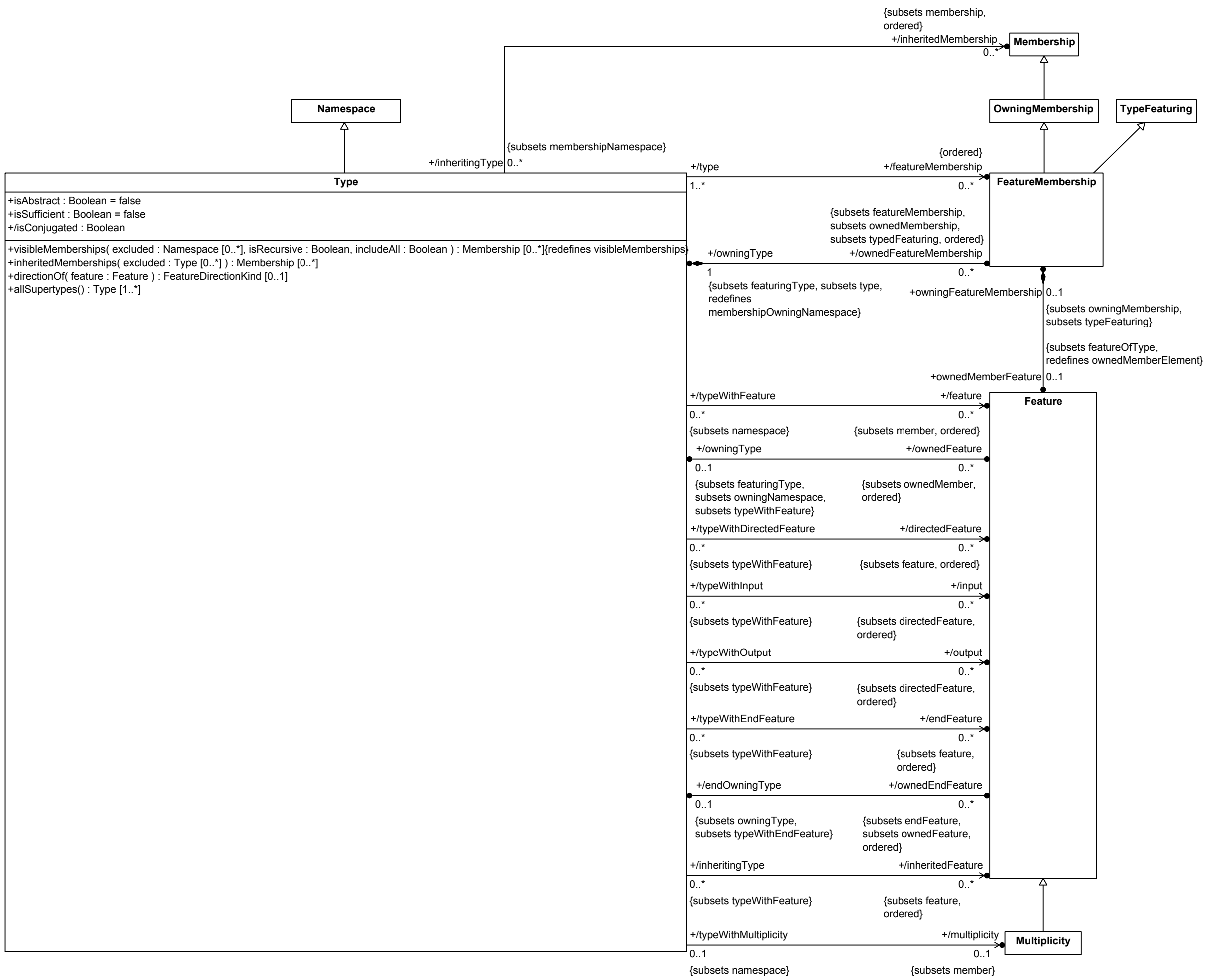

Figure 9. Types

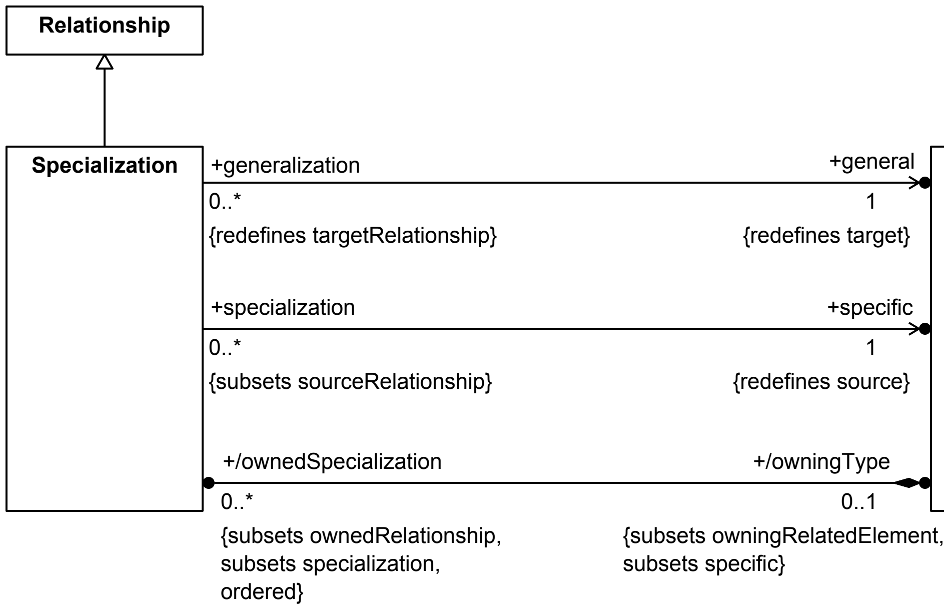

Figure 10. Specialization

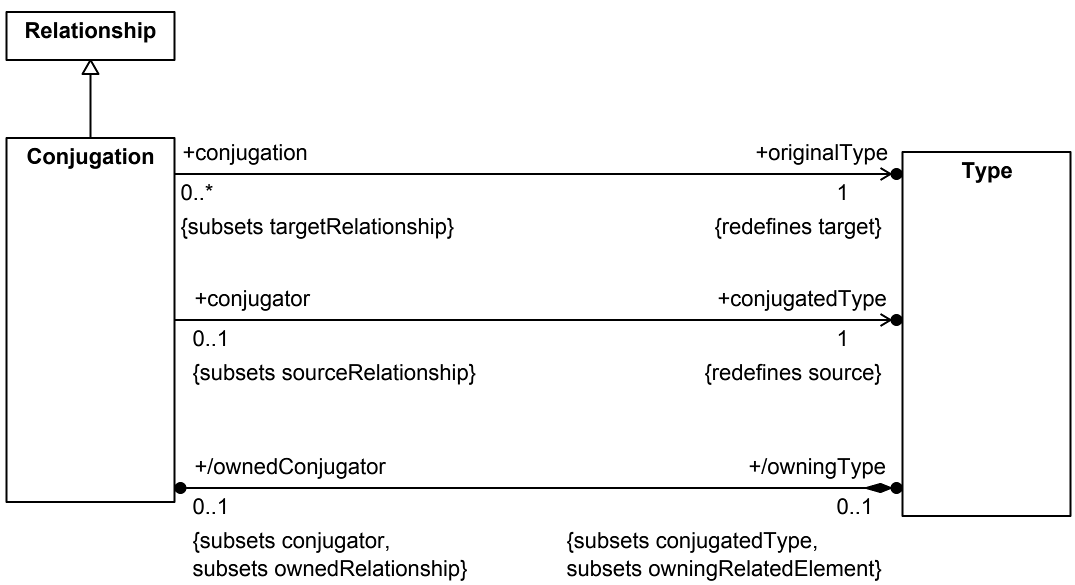

Figure 11. Conjugation

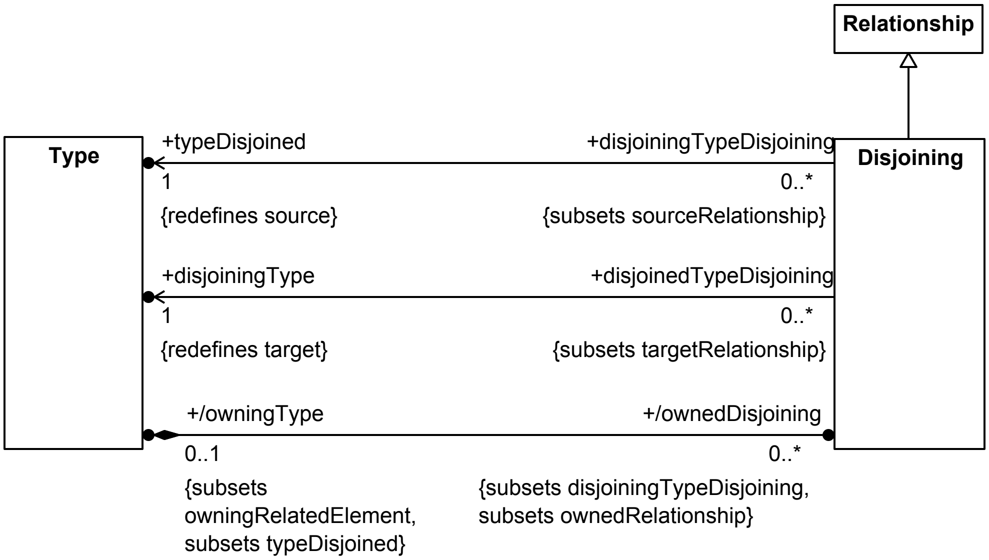

Figure 12. Disjointness

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 `conjugatedType` of this Type that is also its `owningRelatedElement`.

Operations

No operations.

Constraints

None.

7.3.2.3.3 Disjoining

Description

A Disjoining is a Relationship between Types asserted to have interpretations that are not shared (disjoint) between them, identified as `typeDisjoined` and `disjoiningType`. For example, a Classifier for mammals is disjoint from a Classifier for minerals, and a Feature for people's parents is disjoint from a Feature for their children.

General Classes

Relationship

Attributes

disjoiningType : Type {redefines target}

Type asserted to be disjoint with the `typeDisjoined`.

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

A `typeDisjoined` that is also an `owningRelatedElement`.

typeDisjoined : Type {redefines source}

Type asserted to be disjoint with the `disjoiningType`.

Operations

No operations.

Constraints

None.

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

None.

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 an OwningMembership for a Feature in a Type that is also a TypeFeaturing Relationship between the Feature and the Type.

General Classes

TypeFeaturing
OwningMembership

Attributes

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

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

/owningType : Type {subsets type, featuringType, redefines membershipOwningNamespace}

The Type that owns this FeatureMembership.

Operations

No operations.

Constraints

None.

7.3.2.3.6 Specialization

Description

Specialization 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, owningRelatedElement}

The Type that is the `specific` Type of this Specialization 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 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:

- 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 Expression that might vary in their results. If the `typeWithMultiplicity` is a Classifier, the domain of the Multiplicity shall be *Anything*. If the `typeWithMultiplicity` is a Feature, the Multiplicity shall have the same domain as the `typeWithMultiplicity`.

General Classes

Feature

Attributes

None.

Operations

No operations.

Constraints

None.

7.3.2.3.8 Type

Description

A Type is a Namespace 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

Namespace

Attributes

/directedFeature : Feature [0..*] {subsets feature, ordered}

The `features` of this Type that have a non-null `direction`.

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

All `features` related to this Type by EndFeatureMemberships.

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

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

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

The FeatureMemberships for `features` of this Type, which include all `ownedFeatureMemberships` and those `inheritedMemberships` that are FeatureMemberships (but does *not* include any `importedMemberships`).

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

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. These are included in the derived union for the `memberships` of the Type.

/input : Feature [0..*] {subsets directedFeature, ordered}

All `features` related to this Type by FeatureMemberships that have `direction in or inout.`

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 member}

The one `member` (at most) of this Type that is a Multiplicity, which constrains the cardinality of the Type. A `multiplicity` can be owned or inherited. If it is owned, the `multiplicity` must redefine the `multiplicity` (if it has one) of any `general` Type of a `Generalization` of this Type.

/output : Feature [0..*] {subsets directedFeature, ordered}

All `features` related to this Type by FeatureMemberships that have `direction out or inout.`

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

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

/ownedDisjoining : Disjoining [0..*] {subsets ownedRelationship, disjoiningTypeDisjoining}

The `ownedRelationships` of this Type that are Disjoinings, for which the Type is the `typeDisjoined` Type.

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

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

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

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

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

The `ownedMemberships` of this Type that are FeatureMemberships, for which the Type is the `owningType`. Each such FeatureMembership identifies an `ownedFeature` of the Type.

/ownedSpecialization : Specialization [0..*] {subsets specialization, ownedRelationship, ordered}

The `ownedRelationships` of this Type that are Specializations, for which the Type is the `specific` Type.

Operations

allSupertypes() : Type [1..*]

Return all Types related to this Type as supertypes directly or transitively by Generalization Relationships.

```
body: ownedGeneralization->
    closure(general.ownedGeneralization).general->
    including(self)

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.

```
body: if input->includes(feature) and output->includes(feature) then
    FeatureDirectionKind::inout
else if input->includes(feature) then
    FeatureDirectionKind::_'in'
else if output->includes(feature) then
    FeatureDirectionKind::out
else
    null
endif endif endif
```

inheritedMemberships(excluded : Type [0..*]) : Membership [0..*]

Return the inherited Memberships of this Type, excluding those supertypes in the `excluded` set.

visibleMemberships(excluded : Namespace [0..*],isRecursive : Boolean,includeAll : Boolean) : Membership [0..*]

The visible Memberships of a Type include `inheritedMemberships`.

```
body: let visibleInheritedMemberships : Sequence(Membership) =
    inheritedMemberships(excluded)->
        select(includeAll or visibility = VisibilityKind::public) in
self.oclAsType(Namespace).visibleMemberships(excluded, isRecursive, includeAll)->
    union(visibleInheritedMemberships)
```

Constraints

typeOwnedConjugator

[no documentation]

```
let ownedConjugators: Sequence(Conjugator) =
    ownedRelationship->selectByKind(Conjugation) in
    ownedConjugators->size() = 1 and
    ownedConjugator = ownedConjugators->at(1)
```

typeSpecializesAnything

[no documentation]

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

typeOwnedGeneralizations

[no documentation]

```
ownedGeneralization = ownedRelationship->selectByKind(Generalization)->
    select(g | g.special = self)
```

typeOutput

If this Type is conjugated, then its `outputs` are the `inputs` of the `originalType`. Otherwise, its `outputs` are all `features` with FeatureMembership `direction` of `out` or `inout`.

```
output =
    if isConjugated then
        conjugator.originalType.input
    else
        feature->select(direction = out or direction = inout)
    endif
```

typeMultiplicity

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

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

typeInheritedMembership

[no documentation]

```
inheritedMembership = inheritedMemberships(Set{})
```

typeDirectedFeature

[no documentation]

```
directedFeature = feature->select(direction <> null)
```

typeFeatureMembership

The `featureMemberships` of a Type is the union of the `ownedFeatureMemberships` and those `inheritedMemberships` that are FeatureMemberships.

```
featureMembership = ownedMembership->union(
    inheritedMembership->selectByKind(FeatureMembership))
```

typeFeature

The `features` of a Type are the `ownedMemberFeatures` of its `featureMemberships`.

```
feature = featureMembership.ownedMemberFeature
```

typeOwnedFeature

The `ownedFeatures` of a Type are the `ownedMemberFeatures` of its `ownedFeatureMemberships`.

```
ownedFeature = ownedFeatureMembership.ownedMemberFeature
```

typeDisjointType

[no documentation]

```
disjointType = disjoiningTypeDisjoining.disjoiningType
```

typeOwnedFeatureMembership

The `ownedFeatureMemberships` of a Type are its `ownedMemberships` that are FeatureMemberships.

```
ownedFeatureMembership = ownedRelationship->selectByKind(FeatureMembership)
```

typeInput

If this Type is conjugated, then its `inputs` are the `outputs` of the `originalType`. Otherwise, its `inputs` are all `features` with FeatureMembership `direction` of `in` or `inout`.

```
input =
    if isConjugated then
        conjugator.originalType.output
    else
        feature->select(direction = _'in' or direction = inout)
    endif
```

7.3.2.4 Semantics

Required Specializations of Model Library

1. All Types shall directly or indirectly specialize *Base::Anything* (see 8.2.2.1).

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 the Types it specializes.

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

2. No sequences in the interpretation of a Type are in the interpretations of its disjoining Types.

$$\forall t, t_d \in V_T \;\; t_d \in t.\texttt{disjoiningTypeDisjoining.disjoiningType} \Rightarrow (\, (t)^T \cap (t_d)^T = \emptyset \,)$$

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.)

Subclassification

Subclassifications are Specializations that restrict their `specific` and `general` Types to be Classifiers, identifying them as `subclassifier` and `superclassifier`, respectively. The *subclassifiers* (plural, in regular font) of a Classifier in this specification is short for the Classifiers related to it by `subclassification`, and the specializations of those Classifiers, recursively.

7.3.3.2 Concrete Syntax

7.3.3.2.1 Classifiers

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

ClassifierDeclaration (t : Type) =
    ( t.isSufficient ?= 'all' )? Identification(t)
    ( t.ownedRelationship += OwnedMultiplicity )?
    ( SuperclassingPart(t) | ConjugationPart(t) )?
    DisjoiningPart(t)?

SuperclassingPart (t : Type) =
      SPECIALIZES t.ownedRelationship += OwnedSubclassification
      ( ',' t.ownedRelationship += OwnedSubclassification )*
```

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 Specializations defined are specifically Subclassifications. A Classifier is also not required to have any `ownedSubclassifications` explicitly specified. If no explicit Subclassification is given for a Classifier, and the Classifier is not conjugated, then the Classifier is given a default Subclassification to the most general base Classifier *`Anything`* from the *`Base`* model library (see 8.2).

```
classifier Person { // Default superclassifier 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 Subclassification

```
Subclassification : Subclassification =
    ( 'specialization' Identification(this) )?
    'subclassifier' subclassifer = [QualifiedName]
    SPECIALIZES superclassifer = [QualifiedName] ';'

OwnedSubclassification : Subclassification =
    superclassifier = [QualifiedName]
```

A Subclassification Relationship is declared using the keyword **`specialization`**, optionally followed by a `shortName` and/or a `name`. The qualified name of the `subclassifier` is then given after the keyword **`subclassifier`**, followed by the qualified name of the `superclassifier` after the keyword **`specializes`**. The symbol `:>` can be used interchangeably with the keyword **`specializes`**.

```
specialization Super subclassifier A specializes B;
specialization subclassifier B :> A;
```

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

```
subclassifer C specializes A;
subclassifier C specializes B;
```

An `ownedSubclassiciation` 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 `superclassifer` 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

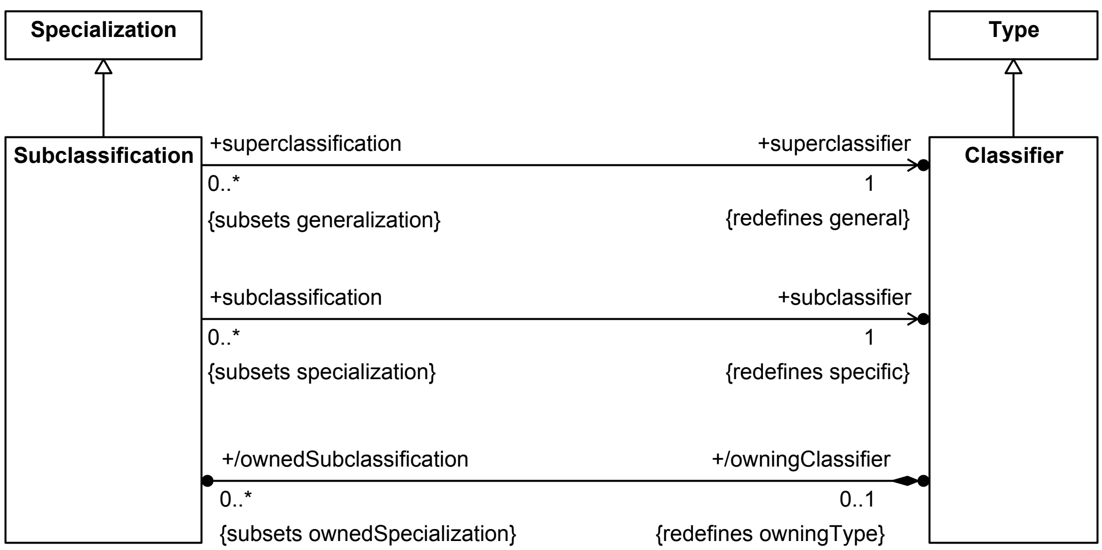

Figure 13. Classifiers

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

/ownedSubclassification : Subclassification [0..*] {subsets ownedSpecialization}

The `ownedSpecializations` of this Classifier that are Subclassifications, for which this Classifier is the `subclassifier`.

Operations

No operations.

Constraints

classifierOwnedSuperclassings

[no documentation]

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

classiferMultiplicityDomain

If a Classifier has a `multiplicity`, then the `multiplicity` shall have no `featuringTypes` (meaning that its domain is implicitly *Base::Anything*).

```
multiplicity <> null implies multiplicity.featuringType->isEmpty()
```

7.3.3.3.3 Subclassification

Description

Subclassification is Specialization 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

Specialization

Attributes

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

The Classfier that owns this Subclassification relationship, which must also be its `subclassifier`.

subclassifier : Classifier {redefines specific}

The more specific Classifier in this Subclassification.

superclassifier : Classifier {redefines general}

The more general Classifier in this Subclassification.

Operations

No operations.

Constraints

None.

7.3.3.4 Semantics

Required Specializations of Model Library

See 7.3.2.4.

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.

   $$(Anything)^T = S$$

7.3.4 Features

7.3.4.1 Features Overview

Features

Features are Types that classify how things in the modeled universe are related, including by chains of relationships. Relations between things can also 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*). Instances in the domain of a Feature are sometimes informally said to "have values" that are instances of 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.

Type Featuring

TypeFeaturing is a Relationship between a Feature and a Type, identified as a Feature's `featuringType` (see above about `featuringType`). TypeFeaturings can owned by a Type, identified as its `ownedTypeFeaturings`. A FeatureMembership is a kind of TypeFeaturing that also makes the Feature a `member` of the `featuringType` (see 7.3.2).

Feature Typing

FeatureTyping is a Specialization between a Feature and a Type, which is identified as a `type` of the Feature (see first paragraph above about `type`). FeatureTyping 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).

FeatureTypings can be owned by their `typedFeature`, identified as one of its `ownedTypings`. 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 Specialization that restricts its `specific` and `general` Types to be Features, identifying them as `subsettingFeature` and `subsettedFeature`, respectively. This means the things identified by (values of) the `subsettingFeature` are also identified by `subsettedFeature` on each instance (separately) of the domain of the `subsettingFeature`. The `subsettingFeature` can restrict any aspect of the `subsettedFeature`, such as the (co)domain and multiplicity. Subsetting can form cycles of Features, which means the extents of the Features are the same, like any Specialization, but a Classifier in the cycle will prevent all the Features and Classifiers in it from having any instances (see Generalization, and Classifiers and Features, in 7.3.2.1).

Redefinition

Redefinition is a Subsetting that requires the things identified by (values of) the `redefiningFeature` and the `redefinedFeature` (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 (directly or 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.

Feature Direction

The `direction` of a Feature specifies what is allowed to change their values on instances of its domain:

- The instance itself (`direction=out`). These features identify things output by an instance.
- Other things "outside" of it (`direction=in`). These parameters identify things inputs to an instance.
- Or both (`direction=inout`).

Feature Chaining

FeatureChaining is a Relationship between one Feature and another, identified as `featureChained` and `chainingFeature`, but the meaning for the first one (`featureChained`) depends on all the other Features it is related to via FeatureChaining, which it identifies in order (a "chain") by `chainingFeature`. The values of a Feature with `chainingFeatures` are the same as values of the last Feature in the chain, which can be found by starting with the values of the first Feature (for each instance of the original Feature's domain), then on each of those to the values of the second Feature in `chainingFeatures`, and so on, to values of the last Feature. The Features related to a Feature by a FeatureChaining are identified as its `chainingFeatures`.

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 has 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 Specialization 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`, in which case they are "nested" features. In this case, the sequences will be longer than 2.

7.3.4.2 Concrete Syntax

7.3.4.2.1 Features

```
FeaturePrefix (f : Feature) =
     ( f.direction = FeatureDirection )?
     ( isAbstract ?= 'abstract' )?
     ( isComposite ?= 'composite' | isPortion ?= 'portion' )?
     ( isReadOnly ?= 'readonly' )?
     ( isDerived ?= 'derived' )?
     ( isEnd ?= 'end' )?

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

Feature : Feature =
    FeaturePrefix
   'feature'? FeatureDeclaration(this)
    ValuePart(this)? TypeBody(this)

FeatureDeclaration (f : Feature) =
     ( f.isSufficient ?= 'all' )? Identification(f)
     ( FeatureSpecializationPart(f) | ConjugationPart(f) )?
    ChainingPart(f)?
    DisjoiningPart(f)?
    TypeFeaturingPart(f)?

TypeFeaturingPart (f : Feature) =
    'featured' 'by' f.ownedRelatioship += OwnedTypeFeaturing
     ( ',' f.ownedTypeFeaturing += OwnedTypeFeaturing )*

ChainingPart (f: Feature) =
    'chains' FeatureChain(f)

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

MultiplicityPart (f : Feature) =
      f.ownedRelationship += OwnedMultiplicity
     | ( f.ownedRelationship += OwnedMultiplicity )?
       ( 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.ownedRelationship += OwnedFeatureTyping )*

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

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

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

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

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

The notation for a Feature is similar to the generic notation for a Type (see 7.3.2.2.1), except using the keyword **feature** rather than **type**. Further, a Feature can have any of three kinds of Specialization: FeatureTyping (see 7.3.4.2.3), Subsetting (see 7.3.4.2.4) and Redefinition (see 7.3.4.2.5). In general, clauses for the different kinds of Specialization 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 of the Specialization clauses in the Feature declaration (but, in all cases, only one multiplicity may be specified). In particular, this allows a notation style for multiplicity consistent with that used in previous modeling languages (such as [UML]). It is also useful when redefining a Feature without giving an explicit name (see 7.3.4.2.5).

```
feature parent[2] : Person;
feature mother : Person[1] :> parent;
feature redefines children[0];
```

In addition to, or instead of, an explicit `multiplicity`, a Feature declaration can include either or both of the following keywords (in either order):

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

There are a number of additional properties of a Feature that can be flagged by adding specific keywords to its declaration. If present these are always specified in the following order, before the keyword **feature**:

1. **in**, **out**, **inout** – Specifies that the Feature has the indicated `direction`.
2. **abstract** – Specifies `isAbstract = true`.

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

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

The keyword **end** is used to set `isEnd = true`, so that the Feature is declared to be an `endFeature`. Any kind of Type can have `endFeatures`, but they are mostly used in Associations (see 7.4.4) and Connectors (see 7.4.5).

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

The `ownedTypings` of a Feature determine its `featuringTypes`. Such `ownedTypings` can be declared for a Feature by including a list of `featuringTypes` in the declaration of the Feature, preceded by the keyword **featured by** and following any `ownedSpecializations`

```
classifier Vehicle;
classifier PoweredComponent;
feature engine : Engine featured by Vehicle, PoweredComponent;
```

7.3.4.2.2 Type Featuring

```
TypeFeaturing : TypeFeaturing =
    'featuring' ( Identification(this) 'of')?
    featureOfType = [QualifiedName]
    'by' featuringType = [QualifiedName] ';'

OwnedTypeFeaturing : TypeFeaturing =
    featuringType = [QualifiedName]
```

A TypeFeaturing Relationship is declared using the keyword **featuring**, optionally followed by a `shortName` and/or a `name` and the keyword **of**. The qualified name of the `featureOfType` is then given, followed by the qualified name of the `featuringType` after the keyword **featured by**.

```
featuring f1 of f featured by A;
featuring g featured by B;
```

An `ownedTypeFeaturing` is defined as part of the declaration of the Feature, rather than in a separate declaration, by including the qualified name of the `featuringType` in a list after the keyword **featured by**.

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

7.3.4.2.3 Feature Typing

```
FeatureTyping : FeatureTyping =
    ( 'specialization' Identification(this) )?
    'typing' typedFeature = [QualifiedName]
    TYPED_BY GeneralType(this) ';'

OwnedFeatureTyping : FeatureTyping =
    GeneralType(this)
```

A FeatureTyping Relationship is declared using the keyword **`specialization`**, optionally followed by a `shortName` 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`, or a Feature chain (see 7.3.4.2.6, after the keyword **`typed by`**. The symbol `:` can be used interchangeably with the keyword **`typed by`**.

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

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

```
typing f typed by B;
typing g : A;
```

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

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

7.3.4.2.4 Subsetting

```
Subsetting : Subsetting =
    ( 'specialization' Identification(this, m) )?
    'subset' SpecificType(this)
    SUBSETS GeneralType(this) ';'

OwnedSubsetting : Subsetting =
    GeneralType(this)
```

A Subsetting Relationship is declared using the keyword **`specialization`**, optionally followed by a `shortName` and/or a `name`. The qualified name of the `subsettingFeature`, or a Feature chain (see 7.3.4.2.6), is then given after the keyword **`subset`**, followed by the qualified name of the `subsettedFeature`, or a Feature chain, after the keyword **`subsets`**. The symbol `:>` can be used interchangeably with the keyword **`subsets`**.

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

If no `shortName` or `name` is given, then the keyword **`specialization`** 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 or Feature chain 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.
}
```

7.3.4.2.5 Redefinition

```
Redefinition : Redefinition =
    ( 'specialization' Identification(this) )?
    'redefinition' SpecificType(this)
    REDEFINES GeneralType(this) ';'

OwnedRedefinition : Redefinition =
    GeneralType(this)
```

A Redefinition Relationship is declared using the keyword **specialization**, optionally followed by a shortName and/or a name. The qualified name of the redefiningFeature, or a Feature chain (see 7.3.4.2.6), is then given after the keyword **redefinition**, followed by the qualified name of the redefinedFeature, or a Feature chain, after the keyword **redefines**. The symbol :>> can be used interchangeably with the keyword **redefines**.

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

If no shortName or name is given, then the keyword **specialization** 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 or Feature chain 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.
    // Legal even though "identifier" is not inherited.
    feature vin redefines identifier;
}
```

If a `name` is not given in the declaration of a Feature with an `ownedRedefinition`, then, its `effectiveName` (which then determines the `ownedMemberName` used in name resolution – see 7.2.4.2.4) is the same as the `effectiveName` of the `redefiningFeature` of its first `ownedRedefinition` (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 7.3.4.2.4) also apply to a `redefiningFeature`.

7.3.4.2.6 Feature Chaining

```
OwnedFeatureChain : Feature =
    FeatureChain(this)

FeatureChain (f : Feature ) =
    f.ownedRelationship += OwnedFeatureChaining
    ( '.' f.ownedRelationship += OwnedFeatureChaining )+

OwnedFeatureChaining : FeatureChaining =
    chainingFeature = [QualifiedName]
```

A Feature chain is a sequence of two or more qualified names separated by dot (`.`) symbols. Each qualified name in a Feature chain shall resolve to a Feature. The first qualified name in a Feature chain shall be resolved in the local Namespace as usual (see 7.2.4.2.4). Subsequent qualified names shall then be resolved using the previously resolved Feature as the context Namespace, but considering only visible Memberships.

The Feature chain notation is used to specify a list of `chainingFeatures` of a Feature, as given by the resolution of the qualified names in the chain, in order. This notation can be placed after the keyword **chains** in the declaration of the Feature, appearing after any specialization or conjugation part, but before any disjoining or type featuring part (see also 7.3.4.2.1).

```
feature cousins chains parents.siblings.children;
```

The `featuringTypes` of the Feature with `chainingFeatures` are implicitly considered to include the `featuringTypes` of the first `chainingFeature`. Similarly, the types of the Feature are implicitly considered to be include the types of the last `chainingFeature`.

The Feature chain notation may also be used for the following:

1. As the `general` Type in the declaration of a Specialization (see 7.3.2.2.2).
2. As the `subsettedFeature` in the declaration of a Subsetting (see 7.3.4.2.4).
3. As the `redefinedFeature` in the declaration of a Redefinition (see 7.3.4.2.5).
4. As the `relatedFeature` of a `connectorEnd` (see 7.4.5.2.1).

In this case, the target of the Relationship being declared (which is some kind of Specialization in all the above cases) shall be a Feature with the a list of `chainingFeatures` as specified by the Feature chain, and that Feature shall also be an `ownedRelatedElement` of the Relationship.

```
feature uncles subsets parents.siblings;
feature cousins redefines parents.siblings.children;
connector vehicle.wheelAssembly.wheels to vehicle.road;
```

Note. A similar dot notation is also used for the related concept of a FeaureChainExpression (see 7.4.8.2.2). However, it always syntactically unambiguous as to whether the notation should be parsed as a plain Feature chain or as a FeatureChainExpression.

7.3.4.3 Abstract Syntax

7.3.4.3.1 Overview

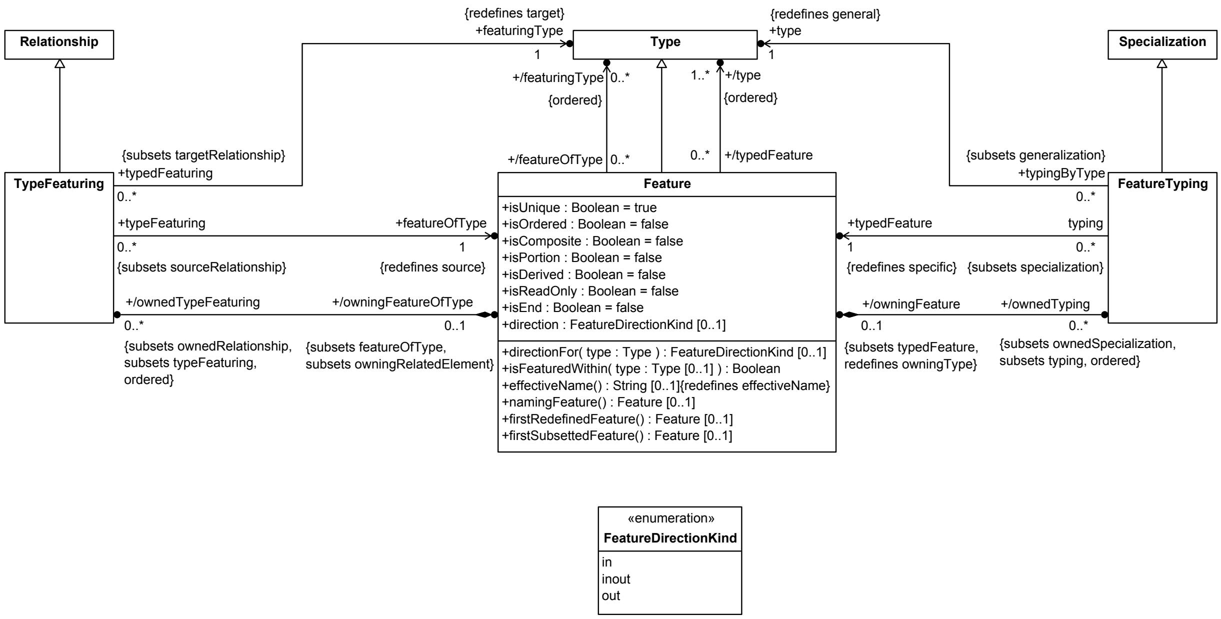

Figure 14. Features

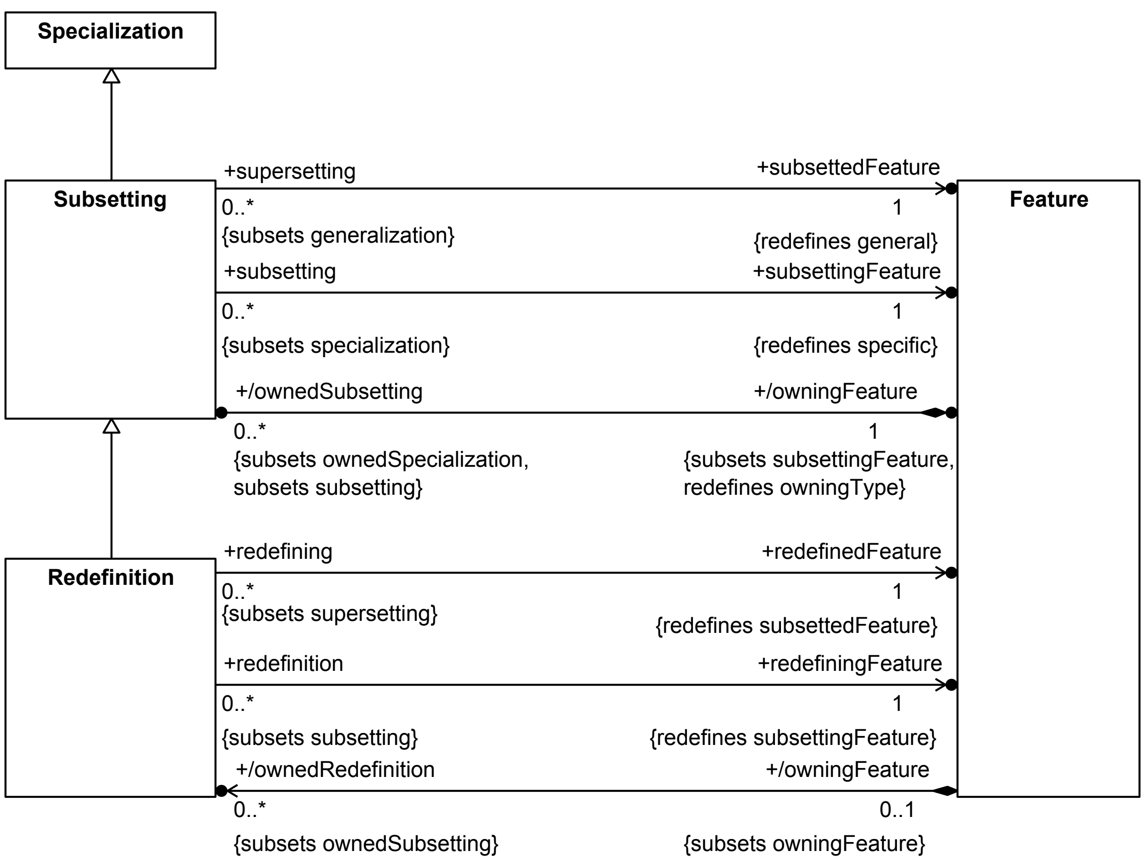

Figure 15. Subsetting

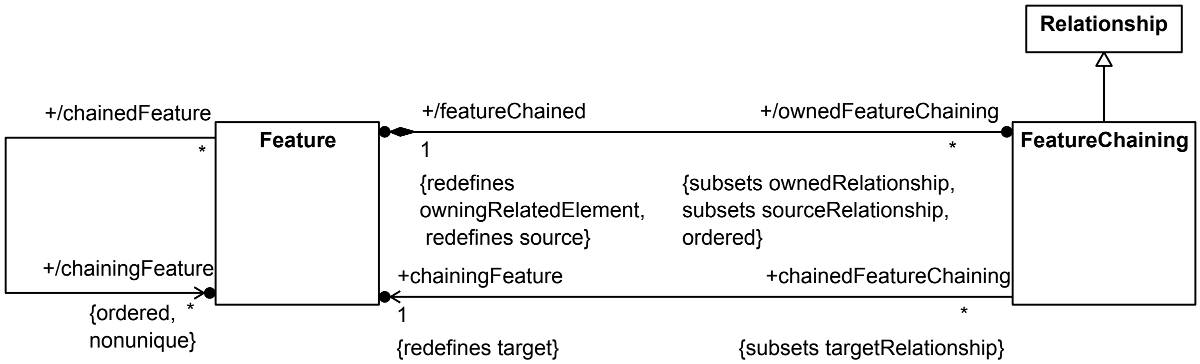

Figure 16. Feature Chaining

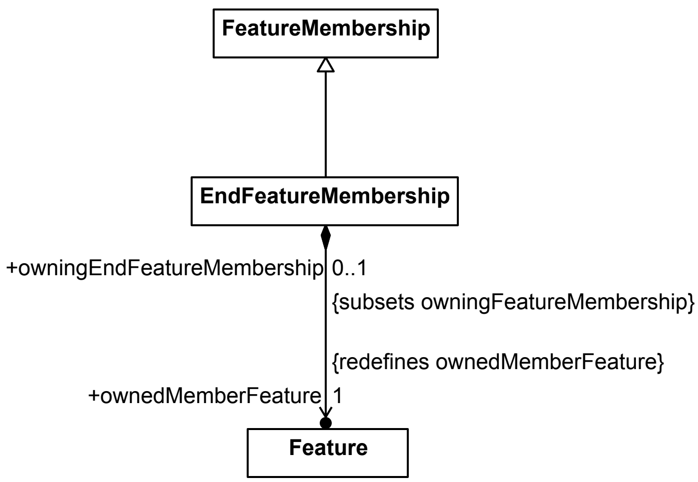

Figure 17. End Feature Membership

7.3.4.3.2 EndFeatureMembership

Description

EndFeatureMembership is a FeatureMembership that requires its `memberFeature` be owned and have `isEnd = true`.

General Classes

FeatureMembership

Attributes

ownedMemberFeature : Feature {redefines ownedMemberFeature}

Operations

No operations.

Constraints

endFeatureMembershipIsEnd

The `ownedMemberFeature` of an EndFeatureMembership must be an end Feature.

```
ownedMemberFeature.isEnd
```

7.3.4.3.3 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*), treating (co)domains as sets of sequences. The domain of Features that do not have any `featuringTypes` is the same as if it were the library Type 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.

The values of a Feature with `chainingFeatures` are the same as values of the last Feature in the chain, which can be found by starting with values of the first Feature, then from those values to values of the second feature, and so on, to values of the last feature.

General Classes

Type

Attributes

/chainingFeature : Feature [0..*] {ordered, nonunique}

The Features that are chained together to determine the values of this Feature, derived from the `chainingFeatures` of the `ownedFeatureChainings` of this Feature, in the same order. The values of a Feature with chainingFeatures are the same as values of the last Feature in the chain, which can be found by starting with the values of the first Feature (for each instance of the original Feature's domain), then on each of those to the values of the second Feature in chainingFeatures, and so on, to values of the last Feature. The Features related to a Feature by a FeatureChaining are identified as its chainingFeatures.

direction : FeatureDirectionKind [0..1]

Determines how values of this Feature are determined or used (see FeatureDirectionKind).

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

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

/featuringType : Type [0..*] {ordered}

Types that feature this Feature, such that any instance in the domain of the Feature must be classified by all of these Types, including at least all the `featuringTypes` of its `ownedTypeFeaturings`.

isComposite : Boolean

Whether the Feature is a composite `feature` of its `featuringType`. If so, the values of the Feature cannot exist after the instance of the `featuringType` no longer does.

.

isDerived : Boolean

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

isEnd : Boolean

Whether or not the this Feature is an end Feature, requiring a different interpretation of the `multiplicity` of the Feature.

An end Feature 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 end Feature, 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.

isOrdered : Boolean

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

isPortion : Boolean

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

isReadOnly : Boolean

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

isUnique : Boolean

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

/ownedFeatureChaining : FeatureChaining [0..*] {subsets sourceRelationship, ownedRelationship, ordered}

The FeatureChainings that are among the `ownedRelationships of this Feature (identify their featureChained also as an owningRelatedElement).`

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

The `ownedSubsettings` of this Feature that are Redefinitions, for which the Feature is the `redefiningFeature`.

/ownedSubsetting : Subsetting [0..*] {subsets ownedSpecialization, subsetting}

The `ownedGeneralizations` of this Feature that are Subsettings, for which the Feature is the `subsettingFeature`.

/ownedTypeFeaturing : TypeFeaturing [0..*] {subsets ownedRelationship, typeFeaturing, ordered}

The `ownedRelationships` of this Feature that are TypeFeaturings, for which the Feature is the `featureOfType`.

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

The `ownedGeneralizations` of this Feature that are FeatureTypings, for which the Feature is the `typedFeature`.

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

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

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

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

/type : Type [1..*] {ordered}

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

Operations

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

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

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

effectiveName() : String [0..1]

If a Feature has no `name`, then its effective name is given by the effective name of the Feature returned by `namingFeature`, if any.

```
body: if name <> null then
    name
else
    let namingFeature : Feature = namingFeature() in
    if namingFeature = null then
        null
    else
        namingFeature.effectiveName()
    endif
endif
```

firstRedefinedFeature() : Feature [0..1]

Return the first Feature that is redefined by this Feature, if any.

```
body: let redefinitions : Sequence(Redefinition) = ownedRedefinition in
if redefinitions->isEmpty() then
    null
else
    redefinitions->at(1).redefinedFeature
endif
```

firstSubsettedFeature() : Feature [0..1]

Get the first Feature that is subsetted by this Feature but not redefined, if any.

```
body: let subsettings : Sequence(Subsetting) =
    ownedSubsetting->reject(oclIsKindOf(Redefinition)) in
if subsettings->isEmpty() then
    null
else
    subsettings->at(1).subsettedFeature
endif
```

isFeaturedWithin(type : Type [0..1]) : Boolean

Return whether this Feature has the given `type` as a direct or indirect `featuringType`. If `type` is null, then check if this Feature is implicitly directly or indirectly featured in *Base::Anything*.

```
body: type = null and feature.featuringType->isEmpty() or
    type <> null and feature.featuringType->includes(type) or
    feature.featuringType->exists(t |
        t.oclIsKindOf(Feature) and
        t.oclAsType(Feature).isFeaturedWithin(type))
```

namingFeature() : Feature [0..1]

By default, the naming feature of a Feature is given by its first `redefinedFeature`, if any.

```
body: firstRedefinedFeature()
```

Constraints

featureChainingFeatureNot1

[no documentation]

```
chainingFeatures->size() <> 1
```

featureType

If a Feature has `chainingFeatures`, then its `types` are the same as the last `chainingFeature`. Otherwise its `types` are the union of the `types` of its `ownedTypings` and the `types` of the `subsettedFeatures` of its `ownedSubsettings`, with all redundant supertypes removed.

.

featureOwnedSubsettings

[no documentation]

```
ownedSubsetting = ownedGeneralization->selectByKind(Subsetting)
```

featureOwnedRedefinitions

[no documentation]

```
ownedRedefinition = ownedSubsetting->selectByKind(Redefinition)
```

featureMultiplicityDomain

If a Feature has a `multiplicity`, then the `featuringTypes` of the `multiplicity` must be the same as those of the Feature itself.

```
multiplicity <> null implies multiplicity.featuringType = featuringType
```

featureRequiredSpecialization

A Feature must directly or indirectly specialize `Base::things` from the Kernel Library.

```
allSupertypes()->includes(KernelLibrary::things)
```

featureOwnedTypeFeaturing

[no documentation]

```
ownedTypeFeaturing = ownedRelationship->selectByKind(TypeFeaturing)->
    select(tf | tf.featureOfType = self)
```

featureOwnedTyping

[no documentation]

```
ownedTyping = ownedGeneralization->selectByKind(FeatureTyping)
```

featureChainingFeaturesNotSelf

A Feature cannot be one of its own `chainingFeatures`.

```
chainingFeatures->excludes(self)
```

featureIsEnd

[no documentation]

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

featureIsDerived

[no documentation]

```
chainingfeatureChainings->notEmpty() implies (owningFeatureMembership <> null implies owningFeatureM
```

featureIsComposite

[no documentation]

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

featureOwnedFeatureChaining

The `ownedFeatureChainings` of this Feature are the `ownedRelationships` that are FeatureChainings.

```
ownedFeatureChaining = ownedRelationship->selectByKind(FeatureChaining)
```

featureChainingFeature

The `chainingFeatures` of a Feature are the `chainingFeatures` of its `ownedFeatureChainings`.

```
chainingFeature = ownedFeatureChaining.chainingFeature
```

7.3.4.3.4 FeatureChaining

Description

FeatureChaining is a Relationship that makes its target Feature one of the `chainingFeatures` of its owning Feature.

General Classes

Relationship

Attributes

chainingFeature : Feature {redefines target}

The Feature whose values partly determine values of `featureChained`, as described in `Feature::chainingFeature`.

/featureChained : Feature {redefines source, owningRelatedElement}

The Feature whose values are partly determined by values of the `chainingFeature`, as described in `Feature::chainingFeature`.

Operations

No operations.

Constraints

None.

7.3.4.3.5 FeatureTyping

Description

FeatureTyping is Specialization in which the `specific` Type is a Feature. This means the set of instances of the (specific) `typedFeature` is a subset of the set of instances of the (general) `type`. In the simplest case, the `type` is a Classifier, whereupon the `typedFeature` subset has instances interpreted as sequences ending in things (in the modeled universe) that are instances of the Classifier.

General Classes

Specialization

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

None.

7.3.4.3.6 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

None.

7.3.4.3.7 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

Specialization

Attributes

/owningFeature : Feature {subsets subsettingFeature, 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

None.

7.3.4.3.8 TypeFeaturing

Description

A TypeFeaturing is a Relationship between a Type and a Feature that is featured by that Type. Every instance in the domain of the `featureOfType` must be classified by the `featuringType`. This means that sequences that are classified by the `featureOfType` must have a prefix subsequence that is classified by the `featuringType`.

General Classes

Relationship

Attributes

featureOfType : Feature {redefines source}

The Feature that is featured by the `featuringType`.

featuringType : Type {redefines target}

The Type that features the `featureOfType`.

/owningFeatureOfType : Feature [0..1] {subsets featureOfType, owningRelatedElement}

The Feature that owns this TypeFeaturing and is also the `featureOfType`.

Operations

No operations.

Constraints

None.

7.3.4.4 Semantics

Required Specializations of Model Library

1. All Features shall directly or indirectly specialize `Base::things` (see 8.2.2.6) (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.

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

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 *feature-pair* 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:

- s_0 is the concatenation of the two sequences, in order.
- The first sequence is in the minimal interpretation of all `featuringTypes` of the Feature.
- The second sequence is in the minimal interpretations of all `types` of the Feature.

$$\begin{array}{l}
\forall s_1, s_2 \in S, f \in V_F \;\; feature\text{-}pair(s_1, s_2, f) \equiv \\
\quad \exists s_0 \in S \;\; s_0 \in (f)^T \wedge concat(s_0, s_1, s_2) \;\; \wedge \\
\quad\quad (\forall t_1 \in V_T \;\; t_1 \in f.\texttt{featuringType} \Rightarrow s_1 \in (t_1)^{minT}) \;\; \wedge \\
\quad\quad (\forall t_2 \in V_T \;\; t_2 \in f.\texttt{type} \Rightarrow s_2 \in (t_2)^{minT})
\end{array}$$

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

3. All sequences in an interpretation of a Feature have a non-overlapping head and tail that are feature pairs of the Feature.

$$\forall s_0 \in S, f \in V_F \;\; s_0 \in (f)^T \Rightarrow \exists s_1, s_2 \in S \;\; head(s_1, s0) \wedge tail(s2, s_0) \wedge (length(s_0) \geq length(s_1) + length(s_2)) \wedge feature\text{-}pair(s_1, s_2, f)$$

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

$$\forall f_g, f_s \in V_F \;\; f_g \in f_s.\texttt{redefinedFeature} \Rightarrow (\forall s_1 \in S \;\; (\forall ft_s \in V_T \; ft_s \in f_s.\texttt{featuringType} \Rightarrow s_1 \in (ft_s)^{minT}) \Rightarrow (\forall s_2 \in S \; (feature\text{-}pair(s_1, s_2, f_s) \equiv feature\text{-}pair(s_1, s_2, f_g))))$$

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

$$\forall s_1 \in S, f \in V_F \;\; \#\{s_2 \mid feature\text{-}pair(s_1, s_2, f)\} \in (f.\texttt{multiplicity})^T$$

6. The interpretation of a Feature with a chain is determined by the interpretations of the subchains, see additional predicates below.

$$\forall f \in V_F, cfl \;\; cfl = f.chainingFeature \wedge \texttt{form:Sequence}(cfl) \wedge length(cfl) > 1 \Rightarrow chain\text{-}feature\text{-}n(f, cfl)$$

The interpretations of a Feature (f) derived from a chain of two others (f_1 and f_2) are all the sequences formed from feature pairs of the two others that share the same sequence as second and first in their pairs, respectively.

$$\forall f, f_1, f_2 \;\; chain\text{-}feature\text{-}2(f, f_1, f_2) \;\Rightarrow\; f \in V_F \;\wedge\; f_1 \in V_F \;\wedge\; f_2 \in V_F$$
$$\forall f, f_1, f_2 \;\; chain\text{-}feature\text{-}2(f, f_1, f_2) \equiv (\forall s_d, s_{cd} \in S \;\; feature\text{-}pair(s_d, s_{cd}, f) \equiv \exists s_m \in S \; feature\text{-}pair(s_d, s_m, f_1) \;\wedge\; feature\text{-}pair(s_m, s_{cd}, f_2))$$

The interpretations of a Feature (f) derived from a chain of two or more others (fl, a list of features longer than 1) is the last in a series of features (flc) that are features derived from subchains, starting with the first two Features in fl, (deriving the first Feature in flc), then the first three (deriving the second Feature in flc), and so on, to all the Features in fl (deriving the last feature in flc, which is the original Feature f).

$$\forall f, fl \;\; chain\text{-}feature\text{-}n(f, fl) \Rightarrow f \in V_F \;\wedge\; fl \subseteq V_F \;\wedge\; \texttt{form:Sequence}(fl) \;\wedge\; length(fl) > 1$$
$$\forall f, fl \;\; chain\text{-}feature\text{-}n(f, fl) \equiv \exists flc \; flc \subseteq V_F \;\wedge\; \texttt{form:Sequence} \;\wedge\; length(flc) = length(fl) - 1 \;\wedge\; (\forall i \in Z^+ \;\; i > 1 \;\wedge\; i <= length(fl) \Rightarrow chain\text{-}feature\text{-}2(at(flc, i-1),\; at(fl, i-1),\; at(fl, i))) \;\wedge\; f = at(flc, length(flc))$$

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 include specialized Classifiers for things that can be identified only by their relations to other things (DataTypes) from others that can be distinguished independently of those relations (Classes and Associations between Classifiers), as well as usages of Associations (Connectors). Classes are for things that exist or happen in time and space. They are divided into those for Structure (classifying things that take up a single region of space and time) and Behavior (classifying things that can be spread out in disconnected portions of space). Structures typically limit how things and relations between them might change over time, while Behaviors specify changes within those limits. Structures and Behaviors do not overlap, but Structures can be involved in, perform, and own Behaviors. Behaviors can coordinate other Behaviors via Steps (usages of Behaviors). Specialized behavioral 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). Some Associations are also Structures (Association Structures).

The Kernel adds semantics beyond the Core primarily by specifying how model elements reuse the Kernel Model Library (see Clause 8), rather than depending only on mathematics, as Core does. Reuse of the Kernel library is specified as constraints in the metamodel. The simplest reuse is specialization (direct or indirect), listed at the beginning of the Semantics subclauses in the rest of this clause. For example, Classes must subclass *Object* from the Objects library model, while Features typed by Classes must subset *`objects`*. Similarly, Behaviors must subclass *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 to redefine the `source` and `target` ends of *BinaryLink*.

The above reuse is covered in the Semantics subclauses with example Kernel model patterns translated to semantically equivalent Core patterns, shown in the textual syntax of each. The Kernel textual syntax introduces keywords that translate to patterns of using Core abstract syntax and libraries, acting as syntactic "markers" for modeling patterns tying Kernel to the Core. It is an example of how other modeling languages can be built on KerML.

Domain-specific metamodels and libraries can also reuse the Kernel metamodel and libraries, inheriting the patterns of library reuse above, as well as the mathematical semantics they inherit from Core. This enables domain-specific modelers to use terms and syntax familiar to them and still benefit from automated assistance based on mathematically-defined semantics.

7.4.2 Classification

7.4.2.1 Classification Overview

Classifiers in Kernel are divided into DataTypes, Classes, and Associations. DataTypes and Classes are specified in this subclause, and Associations in 7.4.4.

Data Types

DataTypes are Classifiers that classify *DataValues*, which are things in the universe that can only be distinguished by their relations to other things (see 8.2.2.2), while Classes and Associations classify things that can be distinguished without regard to those relationships. This means DataTypes cannot also be Classes or Associations, or share instances with them. It also means that DataTypes classify things that do not exist in time or space, because these require changing relationships to other things. 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 Scalar Values Kernel Model Library (see 8.18), and any subtypes of those.

Classes

Classes are Classifiers that classify *Occurrences*, which exist in time and space (see 8.4.2.9). Relations between *Occurrences* and other things can change over time (see Portions and Time Slices in 8.4.1 and *LinkObjects* in 8.5.1).

7.4.2.2 Concrete Syntax

7.4.2.2.1 Data Types

```
DataType : DataType =
    ( isAbstract ?= 'abstract' )? 'datatype'
    ClassifierDeclaration(this) 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 them 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 : Class =
    ( isAbstract ?= 'abstract' )? 'class'
    ClassifierDeclaration(this) 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 *Occurrence* from the *Occurences* model library (see 8.4).

Either all of the `types` of a Feature shall be Classes, or none of them 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 *`occurrences`* from the *Occurrences* model library (see 8.4), unless at least one of the types is an AssociationStructure, in which case the default Superclassing shall be as specified in 7.4.4.2.

```
class Situation { // Specializes Occurrences::Occurrence by default.
    feature condition : ConditionCode;
    feature soundAlarm : ScalarValues::Boolean;
}
class SituationStatusMonitor specializes StatusMonitor {
    feature currentSituation[*] : Situation; // Subsets Occurrences::occurrences by default.
}
```

7.4.2.3 Abstract Syntax

7.4.2.3.1 Overview

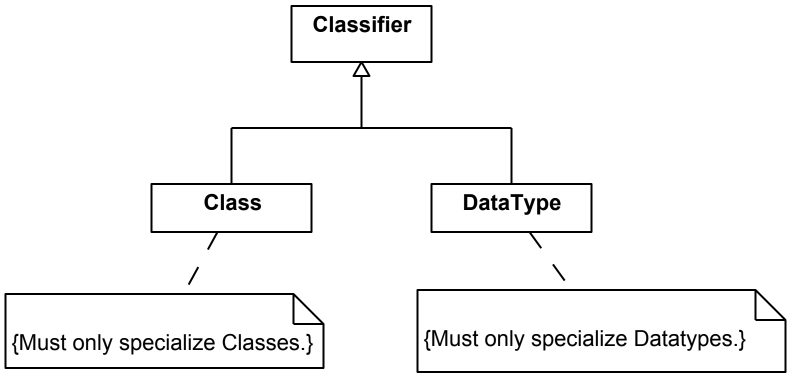

Figure 18. Classification

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

None.

Operations

No operations.

Constraints

classClassifiesOccurrence

[no documentation]

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

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

None.

Operations

No operations.

Constraints

datatypeClassifiesDataValue

[no documentation]

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

7.4.2.4 Semantics

Required Specializations of 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 *Occurrences::Occurrence* (see 8.4.2.9).
4. Features typed by Classes shall (indirectly) subset `Occurrences::occurrences` (see 8.4.2.10).

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, the heads of sequences ending in that thing shall be different than the heads of sequences ending in the other things.

7.4.3 Structures

7.4.3.1 Structure Overview

Structures are Classes that classify *Objects*, which are kinds of *Occurrences* that take up a single region of space and time (see 8.5), as compared to the *Performances* of Behaviors, which can be spread out in disconnected portions of space and time (see 8.6). Structures typically limit how *Objects* and relations between them can change over time, while Behaviors indicate how *Objects* and their relations change. Structures and Behaviors do not overlap, but Structures can own Behaviors, and the *Objects* they classify can be involved in and perform *Performances*.

7.4.3.2 Concrete Syntax

```
Structure : Structure =
    ( isAbstract ?= 'abstract' )? 'struct'
    ClassifierDeclaration(this) TypeBody(this)
```

A Structure is declared as a Classifier (see 7.3.3.2.1), using the keyword **`struct`**. If no `ownedSuperclassing` is explicitly given for the Structure, then it is implicitly given a default Superclassing to the Structure *Object* from the *Objects* model library (see 8.5).

Either all of the `types` of a Feature shall be Structures, or none of them shall be. If they are all Structures, 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 8.5), unless at least one of the types is an AssociationStructure, in which case the default Superclassing shall be as specified in 7.4.4.2.

```
struct Sensor { // Specializes Objects::Object by default.
    feature id : IdNumber;
    feature currentReading : ScalarValues::Real;
    step updateReading { ... } // Performed behavior
}
struct SensorAssembly specializes Assembly {
    composite feature sensors[*] : Sensor; // Subsets Objects::objects by default.
}
```

7.4.3.3 Abstract Syntax

7.4.3.3.1 Overview

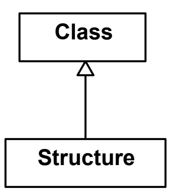

Figure 19. Structures

7.4.3.3.2 Structure

Description

A Structure is a Class of objects in the modeled universe that are primarily structural in nature. While an Object is not itself behavioral, it may be involved in and acted on by Behaviors, and it may be the performer of some of them.

General Classes

Class

Attributes

None.

Operations

No operations.

Constraints

structureClassifiesObject

[no documentation]

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

7.4.3.4 Semantics

Required Specializations of Model Library

1. Structures shall directly or indirectly specialize *Objects::Object* (see 8.5.2.7).
2. Features typed by Structures shall directly or indirectly subset *`Objects::objects`* (see 8.5.2.8).

7.4.4 Associations

7.4.4.1 Associations Overview

Associations are Classifiers that classify *Links* (see 8.3.1) between things in the modeled universe. At least two `ownedFeatures` of an Association must be `endFeatures` (see 7.3.2.1), its `associationEnds`, which identify the things being linked by (at the "ends" of) each Link (exactly one thing per end, which might be the same thing). Associations with exactly two `associationEnds` classify *BinaryLinks* (see 8.3.1), and are called binary Associations. An Association is also a Relationships between the `types` of its `associationEnds`, which might be the same Type, and are identified by its `relatedTypes`. *Links* are between instances of an Association's `relatedTypes`. For binary Associations, the `relatedTypes` are subset into `sourceType` and `targetType`, which might be the same. Associations with more than two `associationEnds` ("n-ary") have only `targetTypes`, no `sourceTypes`. The `features` of Associations that are not `endFeatures` characterize each Link separately from its linked things. AssociationStructures are both Associations and Classes, which classify *LinkObjects*, things that are both *Links* and *Objects* (see 8.5.1).

7.4.4.2 Concrete Syntax

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

AssociationStructure : AssociationStructure =
    ( isAbstract ?= 'abstract' )? 'assoc' 'struct'
    ClassifierDeclaration(this) TypeBody(this)
```

An Association is declared as 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 Association *Link* (otherwise), both of which are from the *Links* library model (see 8.3).

If an Association has a single superclass that is an Association, it may inherit associationEnds from this superclass Association. However, if it declares any owned `associationEnds`, then each of these shall redefine an `associationEnd` of the superclass Association, in order, up to the number of `associationEnds` of the `superClass`. If no redefinition is given explicitly for an owned `associationEnd`, then it shall be considered to implicitly redefine the `associationEnd` at the same position, in order, of the `superClass` Association (including implicit defaults), if any.

```
assoc Ownership { // Specializes Objects::BinaryLink by default.
    feature valuationOnPurchase : 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 has more than one `superclass` that is an Associations, then the Association *must* declare a number of owned `associationEnds` at least equal to the maximum number of `associationEnds` of any of its superclass Associations. Each of these owned `associationEnds` shall then redefine the corresponding `associationEnd` (if any) at the same position, in order, of each of the superclass Associations.

An AssociationStructure is declared like a regular Association, but using the keyword **`assoc struct`**. If no `ownedSuperclassing` is explicitly given for the AssociationStructure, then it is implicitly given a default Superclassing to either the AssociationStructure *BinaryLinkObject* (if it is a binary AssociationStructure) or the AssociationStructyre *LinkObject* (otherwise), both of which are from the *Objects* library model (see 8.5). The same rules on `associationEnds` specified above for Associations also apply to AssociationStructures. An AssociationStructure may specialize an Association that is not an AssociationStructure, but all specializations of an AssociationStructure shall be AssociationStructures.

```
assoc struct ExtendedOwnership specializes Ownership {
     // The values of this feature may change over time.
    feature revaluations[*] ordered : MonetaryValue;
}
```

An Association shall not have any composite `features` if it is not an AssociationStructure. If an AssociationStructure is not binary, then none of its `endFeatures` shall be composite. A binary AssociationStructure shall have at most one composite `endFeature`.

```
assoc struct Assembling {
    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 *Links* model library (see 8.3) or to the Feature *binaryLinkObjects* from the *Objects* model library (see 8.5), if any of the Associations are AssociationStructures. If some of the `types` are Associations, but not binary Associations, then it is given a default Subsetting to the Feature *`links`* from the *Links* model library *see 8.3) or to the Feature *linkObjects* from the *Objects* model library (see 8.5), if any of the Associations are AssociationStructures

7.4.4.3 Abstract Syntax

7.4.4.3.1 Overview

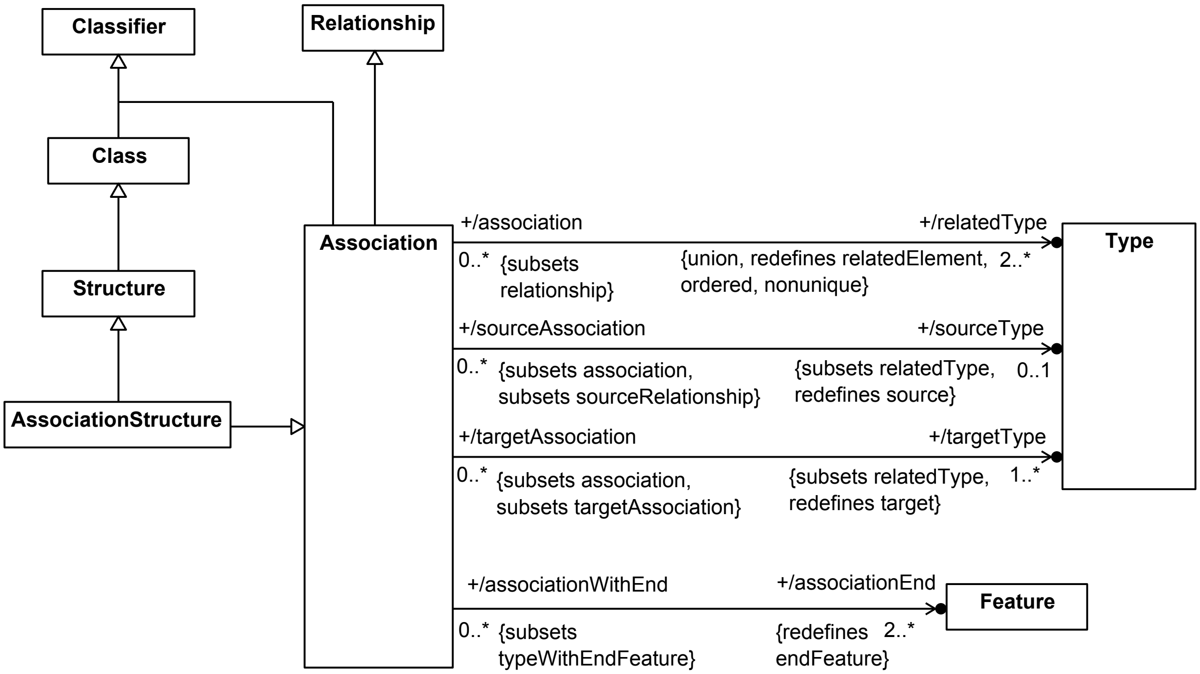

Figure 20. Associations

7.4.4.3.2 Association

Description

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

General Classes

Relationship
Classifier

Attributes

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

The `features` of the Association 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.

/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)
```

associationClassifiesLink

[no documentation]

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

AssociationStructureIntersection

[no documentation]

```
oclIsKindOf(Structure) = oclIsKindOf(AssociationStructure)
```

associationRelatedTypes

[no documentation]

```
relatedTypes = associationEnd.type
```

7.4.4.3.3 AssociationStructure

Description

General Classes

Structure
Association

Attributes

None.

Operations

No operations.

Constraints

associationStructureClassifiesLinkObject

[no documentation]

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

7.4.4.4 Semantics

Required Specializations of Model Library

1. Associations shall directly or indirectly specialize *Links::Link* (see 8.3.2.3).
2. Every `associationEnd` of an Association shall directly or indirectly subset *`Link::participant`*.
3. Associations with exactly two `associationEnds` shall directly or indirectly specialize *Links::BinaryLink* (see 8.3.2.1).
4. Features typed by Associations shall directly or indirectly specialize *`Links::links`* (see 8.3.2.4).
5. Features typed by Associations with exactly two `associationEnds` shall directly or indirectly specialize *`Links::binaryLinks`* (see 8.3.2.2).
6. AssociationStructures shall directly or indirectly specialize *Objects::LinkObject* (see 8.5.2.5).
7. Features typed by AssociationStructures shall directly or indirectly specialize *`Objects::linkObjects`* (see 8.5.2.6).

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

```
classifier A specializes Links::Link {
    end feature e1 subsets Links::Link::participant;
    end feature e2 subsets Links::Link::participant;
    ...
    end feature eN subsets Links::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`*.

Release Note. The special semantics for the multiplicity of end Features is still under discussion.

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 equivalent core 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

```
classifier A specializes Links::BinaryLink {
    end feature e1 redefines Links::BinaryLink::source {
        feature e2 = A::e2(e1);
    }
    end feature e2 redefines Links::BinaryLink::target {
        feature e1 = A::e1(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.

Release Note. The model for navigation across binary Associations is still under discussion.

AssociationStructure Semantics

An AssociationStructure has the same semantics as given above for Associations in general, except that, rather than specializing *Links::Link*, it specializes *Objects::LinkObject,* which in turn specializes *Object,* giving AssociationStuctures the semantics of Structures (see 7.4.3.4) as well as Associations.

7.4.5 Connectors

7.4.5.1 Connectors Overview

Connectors

Connectors are Features that are typed by Associations (see 7.4.4), identifying (having values that are) *Links* (see 8.3.2.3). All Associations typing a Connector shall have the same number of `associationEnds` as the number of owned `endFeatures` of the Connector, its `connectorEnds`. Each `connectorEnd` redefines an `associationEnd` from each of its `types` and subsets a `relatedFeature` of the Connector (exactly one `associationEnd` per `connectorEnd`, and vice-versa, and not more `connectorEnds` than `relatedFeatures`). Connectors typed by binary Associations are called binary Connectors. Connectors are also Relationships between their `relatedFeatures`. For binary Connectors, `relatedFeatures` are subset into `sourceFeature` and `targetFeature`, which might be the same. Connectors with more than two `connectorEnds` ("n-ary") have only `targetFeatures`, no `sourceFeatures`.

Connectors can be thought of as "instance-specific" Associations (usages of Associations), because their values (*Links*) are each limited to linking things identified via `relatedFeatures` on the same instance of the Connector's domain (or by things identified that that instance, recursively, see below). For example, an Association could be used to model an *Engine* driving *Wheels*, and *`type`* a Connector in *Car*. This Connector specifies an *Engine* driving *Wheels* only in the same *Car*, not in another *Car*, as would be allowed with just the Association.

Specifically, the values (*Links*) of a Connector are restricted to those that link things

1. classified by the `types` of its `associationEnds`, regardless of the domain of the Connector.
2. identified by its `relatedFeatures` for the same instance of the domain of the Connector (or by things identified by that instance, recursively).

For example, if the *Wheels* in *Cars* above are taken as parts of their *`driveTrains`*, rather of *Cars* directly, then the *Engine* in each *Car* will drive *Wheels* identified by that *Car*'s `driveTrain`, rather than a Feature of *Car* directly. This requires that each `relatedFeature` of a Connector have some `featuringType` of the Connector as a direct or indirect `featuringType` (where a Feature with no `featuringType` is treated as if the Classifier *Anything* was its `featuringType`). This condition is satisfied if a Connector has an `ownedType` for which its `relatedFeatures` are either direct or `features` reached by chaining. Otherwise, explicit `ownedTypeFeaturing` (see 7.3.4) should be used to ensure that the Connector has a sufficiently general domain.

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 their domain. They are typed by *SelfLink* (which only links things in the modeled universe to themselves, see 8.3.1) and have end multiplicities of exactly 1. This requires a *SelfLink* to exist between each thing identified by the `sourceFeature` and exactly one thing identified by `targetFeature`, and vice-versa.

Since the interpretations of DataTypes are disjoint from those of Classes (see 7.4.2), 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 *DataValues* 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 7.4.10).

Successions

Successions are binary Connectors requiring their `sourceFeature` and `targetFeature` to identify *Occurrences* that are ordered in time. They are typed by the Association *Occurrences::HappensBefore* from the model library (see 8.4.1), which links *Occurrences* that happen completely separately in time, with the Connector's `sourceFeature` being the *`earlierOccurrence`* and the `targetFeature` being the *`laterOccurrence`*.

Successions have properties used in conjunction with *TransitionPerformances::TransitionPerformance* (see 8.10.1):

- `transitionStep` is a Step with `behavior` (typed by) *TransitionPerformance* or a specialization of it, connected to the Succession to the sourceFeature of the Succession, to determine the values (*HappensBefore* Links) of the Succession.
- `triggerSteps` are all the specializations of *TransitionPerformance::`trigger`* on `transitionStep.`
- `guardExpressions` are all the specializations of *TransitionPerformance::`guard`* on `transitionStep.`
- `effectSteps` are all the specializations of *TransitionPerformance::`effect`* on `transitionStep`.

7.4.5.2 Concrete Syntax

7.4.5.2.1 Connectors

```
Connector : Connector =
    FeaturePrefix(this) 'connector'
    ConnectorDeclaration(this) TypeBody(this)

ConnectorDeclaration (c : Connector) : Connector =
    BinaryConnectorDeclaration(c) | NaryConnectorDeclaration(c)

BinaryConnectorDeclaration (c : Connector) : Connector =
    ( FeatureDeclaration(c)? 'from' | c.isSufficient ?= 'all' 'from'? )?
    c.ownedRelationship += ConnectorEndMember 'to'
    c.ownedRelationship += ConnectorEndMember

NaryConnectorDeclaration (c : Connector) : Connector =
    FeatureDeclaration(c)
    ( '(' c.ownedRelationship += ConnectorEndMember ','
        c.ownedRelationship += ConnectorEndMember
        ( ',' c.ownedRelationship += ConnectorEndMember )* ')' )?

ConnectorEndMember : EndFeatureMembership =
    ownedMemberFeature = ConnectorEnd

ConnectorEnd : Feature =
    ( name = Name ':>' )?
    ownedRelationship += OwnedSubsetting
    ( ownedRelationship += OwnedMultiplicity )?
```

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 *Links* model library (see 8.3), if it is a binary Connector, or to the Feature *`links`* from the *Links* model library, if it is not a binary Connector and none of its types are AssociationStructures. If at least one of the types of a Connector is an AssociationStructure, then the default Subsetting is *`linkObjects`* from the *Objects* model library (see 8.5) instead of *`links`*, and, if it is a binary Connector, the default is to subset both *`linkObjects`* and *`binaryLinks`*.

```
// Specializes Objects::LinkObject and Link::BinaryLink by default.
assoc struct Mounting {
    end feature mountingAxle[1] : Axle;
    end feature mountedWheel[2] : Wheel;
}

struct WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;

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

By default, the `connectorEnds` of a Connector 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.

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

Implementation Note. The pairing to `associationEnds` by name is not yet implemented in the pilot implementation. Connector ends are always paired with `associationEnds` in order, even if the named notation is used.

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`**.

```
struct 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.

```
struct 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`.

```
struct 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`.

```
struct 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];
}
```

Note that, if a Connector is an `ownedFeature` of a Type (as above), the context consistency condition for the `relatedFeatures` of a Connector (see 7.4.5.1) requires that these Features also be directly or indirectly nested within the owning Type. The Feature chain dot notation (see 7.3.4.2.6) should be used when connecting so-called "deeply nested" Features.

While the resolution of a Feature path is similar to a qualified name, the Feature path contextualizes the resolution of the final Feature. Thus, for example, while the qualified name *`axle::halfAxles`* statically resolves to *`Axle::halfAxles`*, in the Feature path *`axle.halfAxles`*, *`halfAxles`* is understood to be specifically the Feature as nested in *`axle`*.

```
struct Axle {
    composite feature halfAxles[2] : HalfAxle;
}
struct Wheel {
    composite feature hub : Hub[1];
    composite feature tire : Tire[1];
}
struct WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;

    connector mount : Mounting from axle.halfAxles to wheels.hub;
}
```

7.4.5.2.2 Binding Connectors

```
BindingConnector : BindingConnector =
    FeaturePrefix(this) 'binding'
    BindingConnectorDeclaration(this) TypeBody(this)

BindingConnectorDeclaration (b : BindingConnector) =
    ( FeatureDeclaration(b) 'of' | isSufficient ?= 'all' 'of'? )?
    b.ownedRelationship += ConnectorEndMember '='
    b.ownedRelationship += ConnectorEndMember
```

A BindingConnector is declared as a Feature (see 7.3.4.2) using the keyword **`binding`**. In addition, a BindingConnector declaration gives, after the keyword **`of`**, the qualified names of the two `relatedFeatures` of that are bound by the BindingConnector, separated by the symbol `=`, after the regular Feature declaration part and before the body of the BindingConnector (if any). If no `ownedSubsetting` or `ownedRedefinition` is explicitly given, then the BindingConnector is implicitly given a default Subsetting to the Feature *`selfLinks`* from the *Links* model library (see 8.3). Note that, due to this default subsetting, if no `type` is explicitly given for a BindingConnectgor, then it will implicitly have the `type` *SelfLink* (the type of *`selfLinks`*).

```
struct WheelAssembly {
    composite feature fuelTank {
        out feature fuelFlowOut : Fuel;
    }

    composite feature engine {
        in feature fuelFlowIn : Fuel;
    }

    // Subsets Links::selfLinks by default.
    binding fuelFlowBinding of
```

```
            fuelTank.fuelFlowOut = engine.fuelFlowIn;
    }
```

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

```
    struct WheelAssembly {
        composite feature fuelTank {
            out feature fuelFlowOut : Fuel;
        }

        composite feature engine {
            in feature fuelFlowIn : Fuel;
        }

        binding fuelTank.fuelFlowOut = engine.fuelFlowIn;
    }
```

The `connectorEnds` of a BindingConnector always have multiplicity `1..1`.

7.4.5.2.3 Successions

```
Succession : Succession =
    FeaturePrefix(this) 'succession'
    SuccessionDeclaration(this) TypeBody(this)

SuccessionDeclaration (s : Succession) : Succession =
    ( FeatureDeclaration(s)? 'first' | s.isSufficient ?= 'all' 'first'? )?
    s.ownedRelationship += ConnectorEndMember 'then'
    s.ownedRelationship += 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 8.5). 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;
}
```

7.4.5.3 Abstract Syntax

7.4.5.3.1 Overview

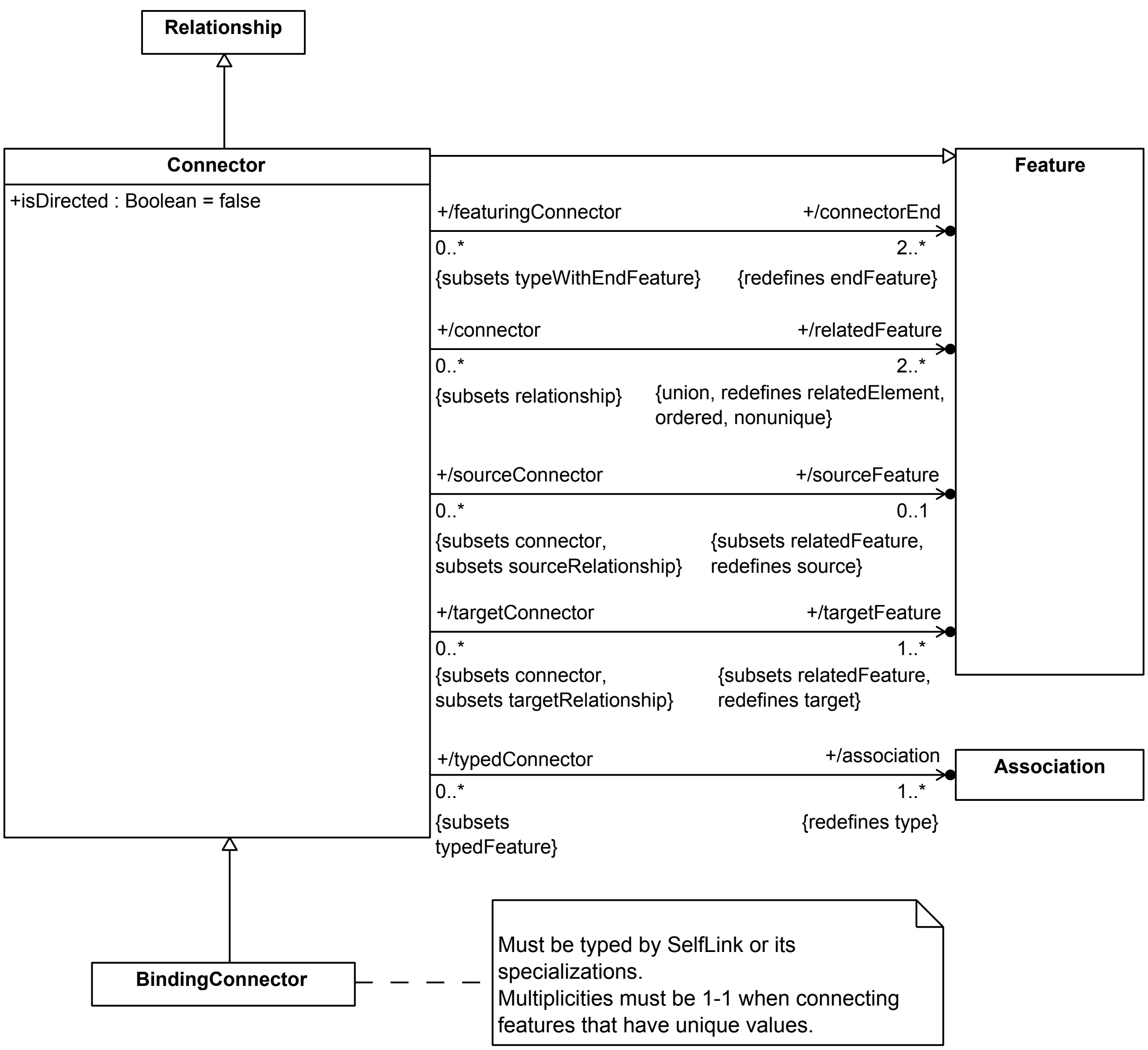

Figure 21. Connectors

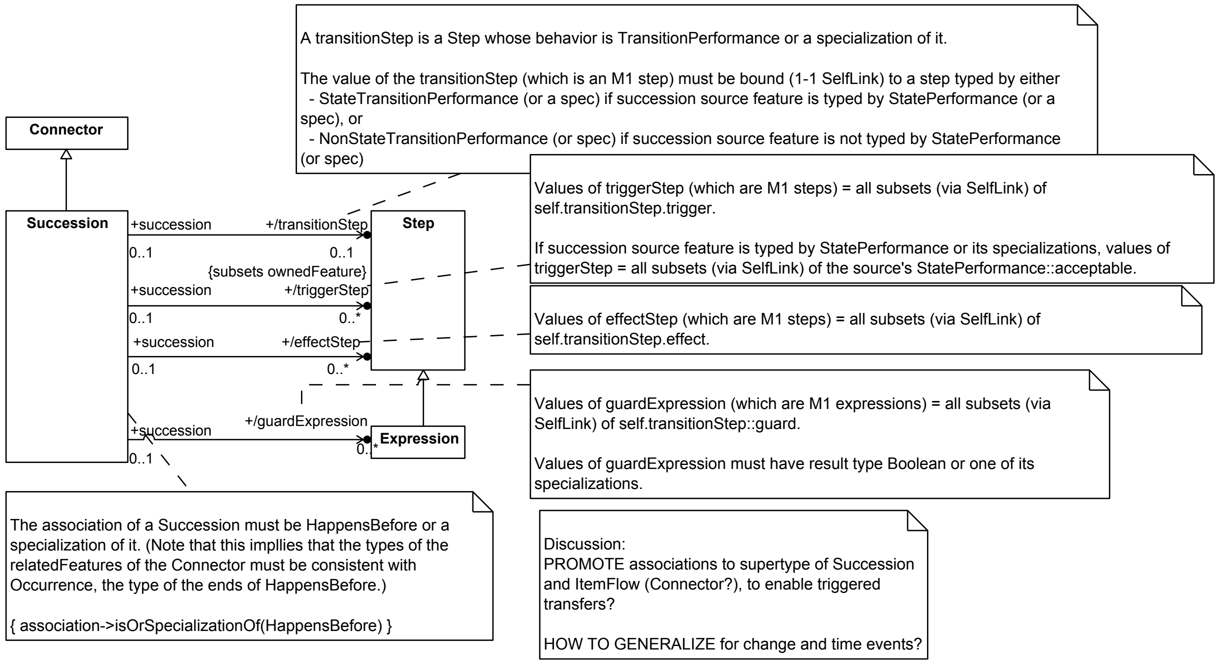

Figure 22. Successions

7.4.5.3.2 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

None.

Operations

No operations.

Constraints

None.

7.4.5.3.3 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..*] {redefines endFeature}

These are the ends of the Connector, which show what Features it relates. The connectorEnds of a Connector are the features of the Connector that redefine the end Features of the Connector association.

isDirected : Boolean

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

/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 end Feature 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 end Feature corresponding to the `targetFeature` must redefine the `target` Feature of the Connector `binaryLinks` from the Kernel Library.

Operations

No operations.

Constraints

connectorEndRedefinition

For each `association` of a Connector, each `associationEnd` must be redefined by a different `connectorEnd` of the Connector.

```
association->forAll(a |
    a.associationEnd->forAll(ae |
        connectorEnd->one(ce |
            ce.ownedRedefinition.redefinedFeature->includes(ae))))
```

connectorTargetFeature

The `targetFeatures` of a Connector are the `relatedFeatures` other than the `sourceFeature`.

```
targetFeature =
    if sourceFeature = null then relatedFeature
    else relatedFeature->excluding(sourceFeature)
    endif
```

connectorConnectorEnd

The `connectorEnds` of a Connector are its `endFeatures`.

```
connectorEnd = feature->select(isEnd)
```

connectorRelatedFeatures

The `relatedFeatures` of a Connector are the subsetted Features of its `connectorEnds`.

```
relatedFeature = connectorEnd.ownedSubsetting.subsettedFeature
```

connectorFeaturingType

Each `relatedFeature` of a Connector must have some `featuringType` of the Connector as a direct or indirect `featuringType` (where a Feature with no `featuringType` is treated as if the Classifier *Base::Anything* was its `featuringType`).

```
relatedFeature->forAll(f |
    if featuringType->isEmpty() then f.isFeaturedWithin(null)
    else featuringType->exists(t | f.isFeaturedWithin(t))
    endif)
```

connectorSourceFeature

If this is a binary Connector, then the `sourceFeature` is the first `relatedFeature`. If this Connector is not binary, then it has no `sourceFeature`.

```
sourceFeature =
    if relatedFeature->size() = 2 then relatedFeature->at(1)
    else null
    endif
```

7.4.5.3.4 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

None.

7.4.5.4 Semantics

Required Specializations of Model Library

1. Connectors shall directly or indirectly specialize *`Links::links`* (see 8.3.2.4), which means they shall be typed by Associations (7.4.4.3.2).
2. Connectors with exactly two `relatedFeatures` shall (indirectly) specialize *`Links::binaryLinks`* (see 8.3.2.2).
3. Connectors with at least one `type` that is an AssociationStructure shall (indirectly) specialize *`Objects::linkObjects`* (see 8.5.2.6).
4. BindingConnectors shall directly or indirectly specialize *`Links::selfLink`* (see 8.3.2.6), which means they shall be typed by (a specialization of) *SelfLink* (see 8.3.2.5).
5. Successions shall (indirectly) specialize *`Occurrences::happensBeforeLinks`* (see 8.4.2.2), which means they shall be typed by (a specialization of) *HappensBefore* (see 8.4.2.1).

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 Links::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 equivalent core model. (If *`A`* is an AssociationStructure, then *`Links::link`* is replaced by *`Objects::LinkObjects`*, above and in the following.)

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 Links::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 c : A from f1 to f2;
```

is semantically equivalent to

```
feature c : A subsets Links::binaryLinks {
    end feature source redefines Objects::binaryLinks::source subsets f1;
    end feature target redefines Objects::binaryLinks::target subsets f2;
}
```

If *`A`* is an AssociationStructure, then the equivalent Feature also subsets *`Objects::linkObjects`*.

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 8.3.2.5). 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 Links::selfLinks {
    end feature thisThing redefines selfLinks::thisThing subsets f1;
    end feature thatThing redefines selfLinks::thatThing 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 8.4.2.1). 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 laterOccurrence
        redefines Occurrences::successions::laterOccurrence subsets f2;
}
```

where *`succession`* is typed by *HappensBefore* and, so, inherits the `endFeatures` *`earlierOccurrence`* and *`laterOccurrence`*.

7.4.6 Behaviors

7.4.6.1 Behaviors Overview

Behaviors

Behaviors are Classes that classify *Performances*, which are kinds of *Occurrences* that can be spread out in disconnected portions of space and time (see 8.6), as compared to *Objects*, which take up a single region of space and time (see 7.4.3 and 8.5). Behaviors can coordinate other Behaviors (see Steps below), specify effects on other things (including their existence and relation to other things), some of which might be accepted as input or provided as output (see Parameters below).

Parameters

Behavior `features` with a non-null `direction` are identified as `parameters` of the Behavior (see Feature Direction in 7.3.4.1). The `direction` of a `parameter` specifies what is allowed to change their values as the Behavior is carried out:

- Performances of the Behavior itself (`direction=out`). These parameters identify things output by a Performance.
- Other things "outside" of it (`direction=in`). These parameters identify things input to a Performance.
- Or both (`direction=inout`).

Steps

Steps are Features typed by Behaviors (`behaviors` of a Step), identifying (having values that are) *Performances* that *HappenDuring* the ones they are Steps of (see 8.4.1). Steps can be connected by Successions to order their values in time via *HappensBefore* (see 7.4.5). They can also be connected by ItemFlows (see 7.4.9), for things flowing between their `parameters` (`out` or `inout` to in or `inout`). Steps can inherit `parameters` of their behaviors or define owned `parameters` to augment or redefine those of their `behaviors`. They can also nest other Steps to augment or redefine `steps` inherited from their `behaviors`.

7.4.6.2 Concrete Syntax

7.4.6.2.1 Behaviors

```
Behavior : Behavior =
    ( isAbstract ?= 'abstract ')? 'behavior'
    BehaviorDeclaration(this) TypeBody(this)

BehaviorDeclaration (b : Behavior) =
    ClassifierDeclaration(b) ParameterList(b)?

ParameterList (t : Type) =
    '(' ( t.ownedRelationship += ParameterMember
        ( ',' t.ownedRelationship += ParameterMember )* )? ')'

ParameterMember : ParameterMembership =
    ownedMemberParameter = ParameterDeclaration

ParameterDeclaration : Feature =
      FeatureParameterDeclaration
    | StepParameterDeclaration
    | ExpressionParameterDeclaration
    | BooleanExpressionParameterDeclaration

FeatureParameterDeclaration : Feature =
    ( direction = FeatureDirection )?
    'feature'? ( f.isSufficient ?= 'all' )? Identification(this)
    ParameterSpecializationPart(this)

StepParameterDeclaration : Step =
    ( direction = FeatureDirection )?
    'step' ( f.isSufficient ?= 'all' )? Identification(this)
    ParameterSpecializationPart(this)

ExpressionParameterDeclaration : Expression =
    ( direction = FeatureDirection )?
    'expr' ( f.isSufficient ?= 'all' )? Identification(this)
    ParameterSpecializationPart(this)

BooleanExpressionParameterDeclaration : BooleanExpression =
    ( direction = FeatureDirection )?
    'bool' ( f.isSufficient ?= 'all' )? Identification(this)
    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 8.6).

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 a Step (see 7.4.6.2.2), Expression (see 7.4.7.2.2) or BooleanExpression (see 7.4.7.2.4) 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**, **end**) 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 ownedSubclassifications whose superclassifiers are Behaviors, then each of the owned parameters of the subclassifier Behavior shall, in order, redefine the parameter at the same position of each of the superclassifier ActionDefinitions. 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 superclassifier Behavior, then the subclassifier Behavior can declare fewer owned parameters than the superclassifier Behavior, inheriting any additional parameters from the superclassifier (which are considered to be ordered after any owned parameters). If there is more than one superclassifier Behavior, then every parameter from every superclassifier must be redefined by an owned parameter of the subclassifier. If every superclassifier parameter is redefined, then the subclassifier 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 superclassifier 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 7.4.6.2.2) 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);
}
```

Like other Type bodies, the body of a Behavior contains a list of declarations of members of the Behavior treated as a Namespace. 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;
    }
```

Any Feature declared in the body of a Behavior with an explicit direction is also considered a parameter of the Behavior. Parameters declared in the body of a Behavior shall be ordered after any parameters given in the declaration of the Behavior, in the lexical order they are declared in the body. They may appear at any location within the body.

```
    behavior TakePicture {
        // The following two features are considered parameters.
        in scene : Scene;
        out picture : Picture;

        binding focus.scene = scene;
        composite step focus : Focus (in scene, out image);
        succession focus then shoot;
        composite flow focus.image to shoot.image;
        composite step shoot : Shoot (in image, out picture);
        binding picture = focus.picture;
    }
```

7.4.6.2.2 Steps

```
Step : Step =
    FeaturePrefix 'step'
    StepDeclaration(this) TypeBody(this)

StepDeclaration (s : Step) =
    FeatureDeclaration(s) ( ValuePart(s) | StepParameterList(s) )?

StepParameterList (t : Type) =
    '(' ( t.ownedRelationship += StepParameterMember
        ( ',' t.ownedRelationship += StepParameterMember )* )? ')'

StepParameterMember : ParameterMembership =
    ownedMemberParameter = StepParameter(this)

StepParameter : Feature =
    ParameterDeclaration 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 8.6). Following the Feature declaration part, a Step declaration can include *either* a FeatureValue (see 7.4.10 or a `parameter` list, declared in the same way as for a Behavior (see 7.4.6.2.1).

```
    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 `superclassifier` Behaviors by a `subclassifier` Behavior (see 7.4.6.2.1).

```
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 7.4.10).

A Step can also have a body, which may have Steps in it. The Step can inherit or redefine Steps from its Behavior `types` or any other Steps it subsets. As in a Behavior body, a Step may also declare parameters within its body (see also 7.4.6.2.1).

```
step takePictureWithAutoFocus : TakePicture {
    in feature unfocusedScene redefines scene;
    step redefines focus : AutoFocus;
    out feature focusedPicture redefines picture;
}
```

7.4.6.3 Abstract Syntax

7.4.6.3.1 Overview

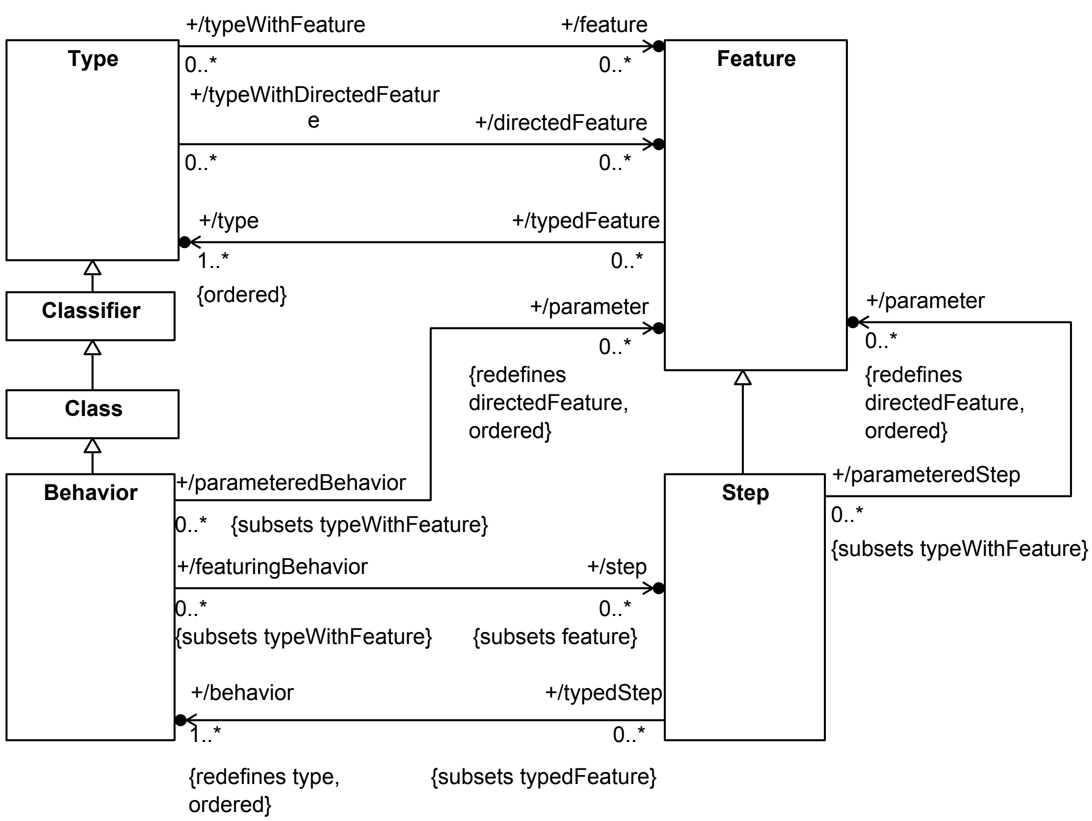

Figure 23. Behaviors

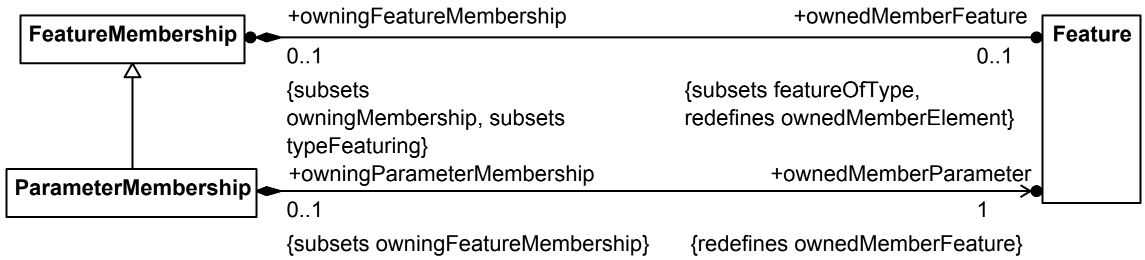

Figure 24. Parameter Memberships

7.4.6.3.2 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

Class

Attributes

/parameter : Feature [0..*] {redefines directedFeature, ordered}

The parameters of this Behavior, which are all its `directedFeatures`, whose values 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

behaviorClassifiesPerformance

[no documentation]

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

7.4.6.3.3 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, ordered}

The Behaviors that type this Step.

/parameter : Feature [0..*] {redefines directedFeature, ordered}

The parameters of this Expression, which are all its `directedFeatures`, whose values are passed into and/or out of a performance of the Behavior.

Operations

No operations.

Constraints

None.

7.4.6.3.4 ParameterMembership

Description

A ParameterMembership is a FeatureMembership that identifies its `memberFeature` as a parameter, which is always owned, and must have a `direction`. A ParameterMembership must be owned by a Behavior or a Step.

General Classes

FeatureMembership

Attributes

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

None.

7.4.6.4 Semantics

Required Specializations of Model Library

1. Behaviors shall directly or indirectly specialize *Performances::Performance* (see 8.6.2.11).
2. Steps shall directly or indirectly specialize *`Performances::performances`* (see 8.6.2.12), 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

```
class 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

```
class 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 Performances::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. The `behaviors` of Steps can have their own Steps, providing for (repeated) refinement of Behaviors by other Behaviors.

7.4.7 Functions

7.4.7.1 Functions Overview

Functions

Functions are Behaviors with all `parameters` having `direction = in` except for exactly one parameter with `direction = out`, known as the `result` parameter.

Functions classify *Evaluations* (see 8.6.2.3), which are kinds of *Performances* that typically produce things (values) identified by their `result` parameter (generally the *`result`* of *Evaluation*). Like all Behaviors, Functions can change things (including those input to them and their result), often referred to as "side effects". A function in the more mathematical sense has no side effects and always produces the same values for its `result` parameter given the same input values. The numerical functions in the Kernel Model Library (see Clause 8) are like mathematical functions.

Expressions

Expressions are Steps (a kind of Feature) typed only by a single Function (their `function`), which means their values are *Evaluations* (see above). They can be `steps` in any Behavior. Functions in particular can designate one of their Expression `steps` as specifying the value of their `result` parameter by a ResultExpressionMembership. The `result` parameter of the designated Expression `step` shall be connected to the `result` parameter of the featuring Function by a BindingConnector (see 7.4.5), ensuring that the two `result` parameters have the same value. This specification sometimes refers to an Expression with a particular *Evaluation* that has a particular value for its `result` parameter as "evaluating to" that value, for short.

Expressions can have their own (nested) `parameters`, to augment or redefine those of their `functions`, including the `result`. They can also own another Expression to specify the value of their `result` parameter. In this case, the owning Expression must connect its `result` parameter with the `result` parameter of its result Expression by a BindingConnector.

See 7.4.8.1 for more about Expressions.

Predicates

Predicates are Functions with their `result` parameter typed by *Boolean* from the Scalar Values library (see 8.18) and `multiplicity` of (exactly) `1`. Predicates determine whether the values of their input parameters meet particular conditions at the time of evaluation, returning (resulting in) `true` if they do, and `false` otherwise. They classify *BooleanEvaluations* (see 8.6.1

Boolean Expressions and Invariants

BooleanExpressions are Expressions whose `function` is a Predicate, and must also have a `result` parameter of type *Boolean*. BooleanExpressions in general might evaluate to `true` at some times and `false` at other times, but Invariants are BooleanExpressions that must always evaluate to either `true` at all times or `false` at all times, as indicated by whether the invariant `isNegated`. By default, an Invariant is asserted to always evaluate to true (`isNegated = false`), while a negated Invariant (`isNegated = true`) is asserted to always evaluate to false.

7.4.7.2 Concrete Syntax

7.4.7.2.1 Functions

```
Function : Function =
    ( isAbstract ?= 'abstract' )? 'function'
    FunctionDeclaration(this) FunctionBody(this)

FunctionDeclaration (f : Function) =
    ClassifierDeclaration(f) ParameterList(f) ReturnParameterPart(f)?

ReturnParameterPart (t : Type) =
    t.ownedRelationship += ReturnParameterMember

ReturnParameterMember : ReturnParameterMembership =
    'return'? ownedMemberParameter = ParameterDeclaration

FunctionBody (t : Type) =
      ';'
    | '{' ( TypeBodyElement(t)
          | ownedRelationship += ReturnFeatureMember )*
          ( t.ownedRelationship += ResultExpressionMember )?
      '}'

ReturnFeatureMember : ReturnParameterMembership =
    MemberPrefix(this) 'return'
    ownedMemberParameter = FeatureElement

ResultExpressionMember : ResultExpressionMembership =
    MemberPrefix(this)
    ownedResultExpression = OwnedExpression
```

A Function is declared as a Behavior (see 7.4.6.2.1), 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, optionally preceded by the keyword **`return`**. 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 8.6). 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 7.4.6.2.1). 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 of a Function is like the body of a Behavior (see 7.4.6.2.1), 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 7.4.8, *not* using the Expression declaration notation from 7.4.7.2.2.

```
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;
}
```

As for a Behavior, any Feature declared in the body of a Function with an explicit direction is also considered a parameter of the Function (see also 7.4.6.2.1). In addition, the result parameter of a Function may be declared in its body by beginning the declaration with the keyword **return** (instead of a direction keyword). In this case, the result parameter shall *not* be included in the declaration of the function signature.

```
function Velocity {
    in v_i : VelocityValue;
    in a : AccelerationValue;
    in dt : TimeValue;
    return v_f : VelocityValue = v_i + a * dt;
}
```

7.4.7.2.2 Expressions

```
Expression : Expression =
    FeaturePrefix(this) 'expr'
    ExpressionDeclaration(this) FunctionBody(this)

ExpressionDeclaration (e : Expression, m : Membership) =
    FeatureDeclaration(e)
    ( ValuePart(e) | StepParameterList(e) ReturnParameterPart(e)? )?
```

An Expression can be declared as a Step (see 7.4.6.2.2) using the keyword **`expr`** (see also 7.4.8.2 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 8.6). Following the Feature declaration part, an Expression declaration can include *either* a FeatureValue (see 7.4.10) *or* a `parameter` list and `result parameter` part, declared in the same way as for a Function (see 7.4.7.2.1).

```
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 7.4.7.2.1).

```
// 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 7.4.10).

An Expression can also have a body which, like a Function body, can specify a result Expression.

```
expr : Dynamics () result : VehicleState {
    vehicleComputation()
}
```

Parameters can also be declared in the body of an Expression, like in a Function body.

```
expr dynamics {
    in state : DynamicState;
    in dt : TimeValue;
    return result : DynamicState;
}
```

7.4.7.2.3 Predicates

```
Predicate : 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 7.4.7.2.1), 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 8.18) and multiplicity `1..1` (see 7.4.11). 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 8.6). 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 7.4.7.2.1).

The body of a Predicate is the same as a Function body (see 7.4.7.2.1). If a result Expression is included, then it shall evaluate to a Boolean `result`.

```
predicate isFull (tank : FuelTank) {
    tank::fuelLevel == tank::maxFuelLevel
}
```

7.4.7.2.4 Boolean Expressions and Invariants

```
BooleanExpression : BooleanExpression =
    FeaturePrefix(this) 'bool'
    ExpressionDeclaration(this) FunctionBody(this)

Invariant : Invariant =
    FeaturePrefix(this) 'inv' ( 'true' | isNegated ?= 'false' )?
    ExpressionDeclaration(this) FunctionBody(this)
```

A BooleanExpression is declared as an Expression (see 7.4.7.2.2), 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 7.4.7.2.3). 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 8.6). 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 7.4.7.2.2).

```
// 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 }
}
```

7.4.7.3 Abstract Syntax

7.4.7.3.1 Overview

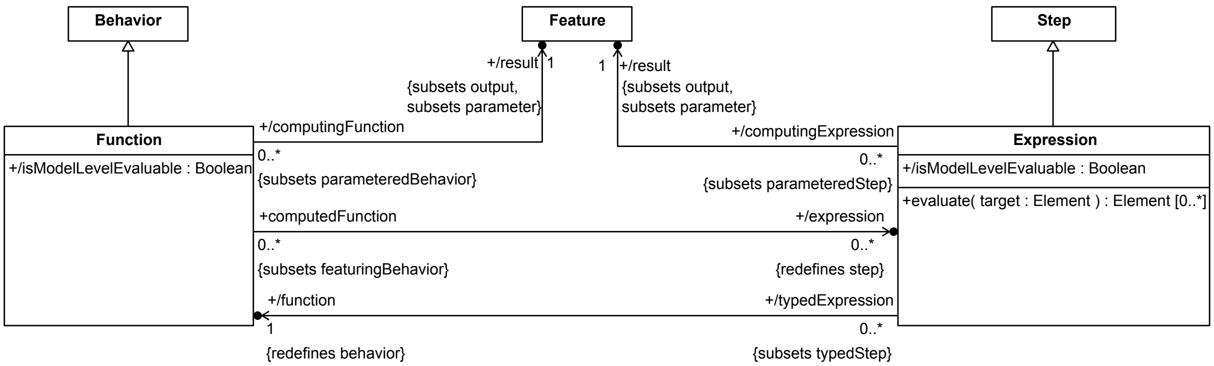

Figure 25. Functions

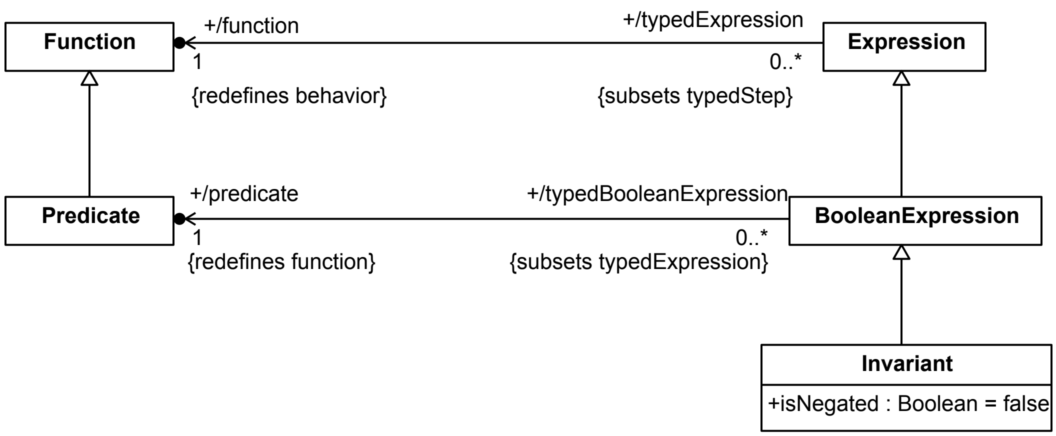

Figure 26. Predicates

Figure 27. Function Memberships

7.4.7.3.2 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

None.

7.4.7.3.3 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.

/isModelLevelEvaluable : Boolean

Whether this Expression meets the constraints necessary to be evaluated at *model level*, that is, using metadata within the model.

/result : Feature {subsets parameter, output}

The `result` parameter of the Expression, derived as the single `parameter` of the Expression with direction `out`. The result of an Expression must either be inherited from its `function` or (directly or indirectly) redefine the `result` parameter of its `function`.

Operations

evaluate(target : Element) : Element [0..*]

If this Expression `isModelLevelEvaluable`, then evaluate it using the `target` as the context Element for resolving Feature names and testing classification. The result is a collection of Elements, each of which must be a LiteralExpression or an AnnotatingFeature.

```
pre: isModelLevelEvaluable
```

Constraints

None.

7.4.7.3.4 Function

Description

A Function is a Behavior that has a single `out parameter` that is identified as its `result`. Any other `parameters` of a Function than the `result` must have direction `in`. A Function represents the performance of a calculation that produces the values of its `result` parameter. This calculation may be decomposed into Expressions that are `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.

/isModelLevelEvaluable : Boolean

Whether this Function can be used as the `function` of a model-level evaluable InvocationExpression.

/result : Feature {subsets parameter, output}

The `result` parameter of the Function, derived as the single `parameter` of the Function with direction `out`.

Operations

No operations.

Constraints

None.

7.4.7.3.5 Invariant

Description

An Invariant is a BooleanExpression that is asserted to have a specific Boolean result value. If `isNegated = false`, then the Invariant must subset, directly or indirectly, the BooleanExpression *trueEvaluations* from the

Kernel library, meaning that the result is asserted to be true. If `isNegated = true`, then the Invariant must subset, directly or indirectly, the BooleanExpression *falseEvaluations* from the Kernel library, meaning that the result is asserted to be false.

General Classes

BooleanExpression

Attributes

isNegated : Boolean

Whether this Invariant is asserted to be false rather than true.

Operations

No operations.

Constraints

None.

7.4.7.3.6 Predicate

Description

A Predicate is a Function whose `result` Parameter has type *Boolean* and multiplicity 1..1.

General Classes

Function

Attributes

None.

Operations

No operations.

Constraints

None.

7.4.7.3.7 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

None.

7.4.7.3.8 ReturnParameterMembership

Description

A ReturnParameterMembership is a ParameterMembership that indicates that the `memberParameter` is the `result` parameter of a Function or Expression. The `direction` of the `memberParameter` must be `out`.

General Classes

ParameterMembership

Attributes

None.

Operations

No operations.

Constraints

None.

7.4.7.4 Semantics

Required Specializations of Model Library

1. Functions shall directly or indirectly specialize *Performances::Evaluation* (see 8.6.2.3).
2. Predicates shall directly or indirectly specialize *Performances::BooleanEvaluation* (see 8.6.2.1).
3. Expressions shall directly or indirectly specialize *`Performances::evaluations`* (see 8.6.2.4), which means they shall be directly or indirectly typed by *Performances::Evaluation*.
4. BooleanExpressions (including Invariants) shall directly or indirectly specialize *`Performances::booleanEvaluations`* (see 8.6.2.2), which means they shall be typed by (a specialization 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 7.4.10).

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 7.4.7.1) 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;
}
```

7.4.8 Expressions

7.4.8.1 Expressions Overview

Expressions

Expressions are Steps (kinds of Features) typed by a single Function, with that Function's *Evaluations* as values (see 8.6.1). See 7.4.7.1 for basic Expressions, including BooleanExpressions and Invariants.

Literal and Null Expressions

LiteralExpressions are Expressions that have the values of their `result` parameter specified as a constant in models by a LiteralExpression's `value` property, ultimately being *DataValues* in the *`result`* of *LiteralEvaluations* classified by the LiteralExpression (see 8.6.1). LiteralInfinities are LiteralExpressions resulting in a number greater than all the integers ("infinity"), but treated like one, notated as `*` (see 8.18.1).

NullExpressions are Expressions that have no values for their `result` parameter in *NullEvaluation*s, which are classified by a NullExpression.

Expression Trees

Expressions are commonly organized into tree structures, with Expressions as the nodes, and the input `parameters` of each Expression connected by BindingConnectors to the `result` parameter of each of its child Expressions in the tree (its `arguments` parameters). KerML includes extensive textual syntax for constructing Expression trees, including traditional operator notations (see 7.4.8.2) 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 7.4.8.3):

- The non-leaf nodes of an Expression tree are InvocationExpressions, a kind of Expression that specifies its inputs (`arguments`, a kind of `ownedFeature`) 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 evaluate to values of a referenced Feature that is not part of the Expression tree, by subsetting the referenced Feature.
  - LiteralExpressions evaluate to the literal value of one of the primitive DataTypes from the Scalar Values model library (see 8.18).
  - NullExpressions evaluate to the empty set (see above).

Body Expressions

An Expression can be the `referent` of a FeatureReferenceExpression in an Expression tree, as above. This enables the Evaluation *`result`* of the `referent` Expression to be taken as the value of an `argument` of an invocation, rather than passing the value of the `result` parameter of the Expression. The Expression Evaluation can be constrained in the context of the performance of the invocation. In particular, if the Expression has `parameters`, these can be bound within the invocation, enabling the Expression to be evaluated within that context.

As a shorthand for doing this, the concrete syntax for an Expression body (as defined in 7.4.7.2.2) can be used as an leaf node in the Expression tree syntax (see 7.4.8.2.3). If this body Expression is used as the `argument` of an InvocationExpression, then the `argument` Expression is bound to the input `parameter` of the InvocationExpression, rather than the `result` of the Expression. This avoids introducing an intermediate FeatureReferenceExpression.

Model-Level Evaluable Expressions

A *model-level evaluable* Expression is an Expression that refers to metadata, which is data about model elements, rather than the things being modeled. Model-level evaluable Expressions can give values to the `metadataFeatures` of an AnnotatingFeature (see 7.4.12) and be used as element filtering conditions in Packages (see 7.4.13). The expressiveness model-level evaluable Expressions is restricted to support this, see below.

All NullExpressions, LiteralExpressions and FeatureReferenceExpressions are model-level evaluable. An InvocationExpression is model-level evaluable if it meets the following conditions:

1. All its `argument` Expressions are model-level evaluable.
2. It has a single `function` that is a library Function listed as being model-level evaluable in Table 6 or Table 8.

In all other cases, an Expression shall be considered to be *not* model-level evaluable.

Rlease Note. The Functions allowed in model-level evaluable expressions may be expanded in the final submission.

7.4.8.2 Concrete Syntax

7.4.8.2.1 Operator Expressions

```
OwnedExpressionReferenceMember : FeatureMembership =
    ownedRelationship += OwnedExpressionReference

OwnedExpressionReference : FeatureReferenceExpression =
    ownedRelationship += OwnedExpressionMember

OwnedExpressionMember : FeatureMembership =
    ownedFeatureMember = OwnedExpression

OwnedExpression : Expression =
      ConditionalExpression
    | BinaryOperatorExpression
    | UnaryOperatorExpression
    | ClassificationExpression
    | ExtentExpression
    | PrimaryExpression

ConditionalExpression : OperatorExpression =
      ownedRelationship += OwnedExpressionMember
      operator = '?'
      ownedRelationship += OwnedExpressionReferenceMember ':'
      ownedRelationship += OwnedExpressionReferenceMember
    | 'if' ownedRelationship += OwnedExpressionMember
      operator = '?'
      ownedRelationship += OwnedExpressionReferenceMember 'else'
      ownedRelationship += OwnedExpressionReferenceMember

ConditionalBinaryOperatorExpression : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = ConditionalBinaryOperator
    ownedRelationship += OwnedExpressionReferenceMember

ConditionalBinaryOperator =
    '??' | '||' | '&&' | 'or' | 'and' | 'implies'

BinaryOperatorExpression : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = BinaryOperator
    ownedRelationship += OwnedExpressionMember

BinaryOperator =
      '|'  | '&' | '^^' | 'xor' | '==' | '!='
    | '..' | '<' | '>'  | '<='  | '>=' | '+'
    | '-'  | '*' | '/'  | '%'   | '^'  | '**'

UnaryOperatorExpression : OperatorExpression =
    operator = UnaryOperator
    ownedRelationship += OwnedExpressionMember

UnaryOperator =
    '+' | '-' | '!' | '~' | 'not'

ClassificationExpression : OperatorExpression =
    ( ownedRelationship += OwnedExpressionMember )?
    operator = ClassificationOperator
    ownedRelationship += TypeReferenceMember
```

```
ClassificationOperator =
    'istype' | 'hastype' | '@' | 'as'

ExtentExpression : OperatorExpression =
    operator = 'all'
    ownedRelationship += TypeReferenceMember

TypeReferenceMember : FeatureMembership =
    ownedMemberFeature = TypeReference

TypeReference : Feature =
    ownedRelationship += ReferenceTyping

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

Operator expressions provide a shorthand notation for InvocationExpressions that invoke a library Function represented as an *operator symbol.* Table 6 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 7.4.8.4 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 or third 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. There are two forms of conditional expressions, both of which place the `?` operator after the first operand. The first form separates the second and third operands with a `:` symbol, while the second form begins with the keyword **`if`** and separates the second and third operands with the keyword **`else`**.

  ```
  x >= 0? x: -x
  if x >= 0? x else -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 (`??`), conditional and (`&&`) and conditional or (`||`) operators all correspond the control Functions (see 8.36) 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). The keywords **`and`**, **`or`** and **`xor`** can be used as synonyms for the `&&`, `||` and `^^` operators, respectively.

  ```
  x + y
  list[i] ?? default
  i > 0 && sensor[i] != null
  sensor == null or sensor.reading > 0
  ```

- *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. The keyword **`not`** can be used as a synonym for the `!` operator.

```
-x
!isOutOfRange(sensor)
not completed
```

- *Classification expression.* The *classification operators* are syntactically similar to binary operators, but, instead of an Expression as their second operand, they take a Type name. The classification operators **istype** and **hastype** test whether the value of their first operand is classified by the named Type (either including or not including subtypes, respectively). The symbol `@` can be used as a synonym for **istype**.

  ```
  sensor istype ThermalSensor
  sensor @ ThermalSensor
  person hastype Administrator
  ```

  The classification operator **as**, known as the *cast operator,* performs an **isType** test of whether each of the values of its first operand is classified by the named Type, and then it selects only those values that pass the test to include in its result. The result values of such a cast expression (if any) are always guarenteed to be instances of the named Type.

  ```
  allSensors as ThermalSensor
  person as Administrator
  ```

  The classification operators may also be used without a first operand, in which case the first operand is implicitly `Anything::self` (see 8.2.2.1). This is useful, in particular, when used as a test within an element filter condition Expression (see 7.4.13.2).

  ```
  istype ThermalSensor
  @ThermalSensor
  hastype Administrator
  as Supervisor
  ```

- *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 7. 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 6. Operator Mapping

Operator	Library Function	Description	Model-Level Evaluable?
all	`BaseFunctions::'all'`	Type extent	No
istype	`BaseFunctions::'istype'`	Is directly or indirectly instance of type	Yes
hastype	`BaseFunctions::'hastype'`	Is directly instance of type	Yes

Operator	Library Function	Description	Model-Level Evaluable?
as	`BaseFunctions::as`	Select instances of type (cast)	Yes
`@`	`BaseFunctions::'@'`	Same as 'istype'	Yes
`==`	`BaseFunctions::'=='`	Equality	Yes
`!=`	`BaseFunctions::'!='`	Inequality	Yes
`=>` **implies**	`DataFunctions::'implies'`	Logical "implication"	Yes
`|`	`DataFunctions::'|'`	Logical "inclusive or"	Yes
`^^` **xor**	`DataFunctions::'^^'`	Logical "exclusive or"	Yes
`&`	`DataFunctions::'&'`	Logical "and"	Yes
`!` **not**	`DataFunctions::'!'`	Logical "not"	Yes
`<`	`DataFunctions::'<'`	Less than	Yes
`>`	`DataFunctions::'>'`	Greater than	Yes
`<=`	`DataFunctions::'<='`	Less than or equal to	Yes
`>=`	`DataFunctions::'>='`	Greater than or equal to	Yes
`+`	`DataFunctions::'+'`	Addition	Yes
`-`	`DataFunctions::'-'`	Subtraction	Yes
`*`	`DataFunctions::'*'`	Multiplication	Yes
`/`	`DataFunctions::'/'`	Division	Yes
`%`	`DataFunctions::'%'`	Remainder	Yes
`^ **`	`DataFunctions::'^'`	Exponentiation	Yes
`..`	`DataFunctions::'..'`	Range construction	Yes
`??`	`ControlFunctions::'??'`	Null coalescing	Yes
`||` **or**	`ControlFunctions::'||'`	Conditional "or"	Yes
`&&` **and**	`ControlFunctions::'&&'`	Conditional "and"	Yes
`?`	`ControlFunctions::'?'`	Conditional test (ternary)	Yes

Table 7. Operator Precedence (highest to lowest)

Unary
all
`+ - ! ~` **not**
Binary
`^ **`
`* / %`
`+ -`

`..`
`< > <= >=`
`istype hastype as` @
`== !=`
`& &&` **`and`**
`^^` **`xor`**
`| ||` **`or`**
`=>` **`implies`**
`??`
Ternary
`?`

7.4.8.2.2 Primary Expressions

```
PrimaryExpression : Expression =
      FeatureChainExpression
    | NonFeatureChainPrimaryExpression

PrimaryExpressionMember : FeatureMembership =
    ownedMemberFeature = PrimaryExpression

NonFeatureChainPrimaryExpression : Expression =
      IndexExpression
    | SequenceExpression
    | SelectExpression
    | CollectExpression
    | FunctionOperationExpression
    | BaseExpression

NonFeatureChainPrimaryExpressionMember : FeatureMembership =
    ownedMemberFeature = NonFeatureChainPrimaryExpression

IndexExpression : OperatorExpression =
    ownedRelationship += PrimaryExpressionMember
    operator = '['
    ownedRelationship += OwnedExpressionMember ']'

SequenceExpression : Expression =
    '(' ( OwnedExpression | SequenceExpressionList ) ','? ')'

SequenceExpressionList : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = ','
    ( ownedRelationship += SequenceExpressionListMember
    | ownedRelationship += OwnedExpressionMember )

SequenceExpressionListMember : FeatureMembership =
    ownedMemberFeature = SequenceExpressionList

FeatureChainExpression : FeatureChainExpression =
    ownedRelationship += NonFeatureChainPrimaryExpressionMember '.'
    ownedRelationship += FeatureChainMember

CollectExpression : CollectExpression =
    ownedRelationship += PrimaryExpressionMember '.'
    ownedRelationship += BodyExpressionMember

SelectExpression : SelectExpression =
    ownedRelationship += PrimaryExpressionMember '.?'
    ownedRelationship += BodyExpressionMember

FunctionOperationExpression : InvocationExpression =
    ownedRelationship += PrimaryExpressionMember '->'
    ownedRelationship += ReferenceTyping
    ( ownedRelationship += BodyExpressionMember
    | ownedRelationship += FunctionReferenceExpressionMember
    | ArgumentList )

BodyExpressionMember : FeatureMembership =
    ownedMemberFeature = BodyExpression
```

```
FunctionExpressionMember : FeatureMembership =
    ownedMemberFeature = FunctionReferenceExpression

FunctionReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FunctionReferenceMember

FunctionReferenceMember : FeatureMembership =
    ownedMemberFeature = FunctionReference

FunctionReference : Expression =
    ownedRelationship += ReferenceTyping

FeatureChainMember : Membership =
      FeatureReferenceMember
    | OwnedFeatureChainMember

OwnedFeatureChainMember : OwningMembership =
    ownedMemberElement = FeatureChain
```

The *primary expressions* provide additional shorthand notations for certain kinds of InvocationExpressions. ~~For those cases in which the InvocationExpression is an OperatorExpression, its `operator` shall be resolved to the appropriate library function as given in~~ Table 8.

Primary expressions include the following:

- *Index expression.* An index expression specifies the invocation of the indexing Function `[` from the *BaseFunctions* library model (see 8.21). The default behavior for this Function is given by the specialization *`SequenceFunctions::'['`*, for which the first operand 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 *`BaseFunctions::'['`* operator may be specialized differently for domain-specific types.

  ```
  sensors[activeSensorIndex]
  ```

- *Sequence expression.* A sequence expression consists of a list of one or more Expressions separated by comma (`,`) symbols, optionally terminated by a final comma, all surrounded by parentheses `(...)`. Such an expression specifies sequential invocations of the sequence concatenation function from the *BaseFunctions* library model (see 8.21). The default behavior for this Function is given by the specialization *`SequenceFunctions::','`*, which concatenates the sequence of values resulting from evaluating its two arguments. With this behavior, a sequence expression concatenates, in order, the results of evaluating all the listed Expressions.

  ```
  (temperatureSensor, windSensor, precipitationSensor)
  ( 1, 3, 5, 7, 11, 13, )
  ```

  A sequence expression with a single constituent Expression simply evaluates to the value of the contained Expression, as would be expected for a parenthesized expression. The empty sequence `()` is not actually a sequence expression, but, rather, an alternative notation for a NullExpression (see 7.4.8.2.3).

  ```
  (highValue + lowValue) / 2
  ```

  Note. Sequences of values are *not* themselves values. Therefore, sequences are "flat", with no element of a sequence itself being a sequence. For example, ((1, 2, 3), 4), (1, (2, 3), 4) and (1, null, (2, 3, 4)) all evaluate to the same sequence of values as (1, 2, 3, 4). To model nested collection values, use the DataTypes from the *Collections* library model (see 8.19).

- *Feature chain expression.* A feature chain expression consists of a primary Expression and a Feature qualified name or a Feature chain (7.3.4.2.6), separated by a dot (`.`) symbol. The referenced Feature is evaluated in the context of each of the result values of the primary Expression, in order. The resulting Feature values are then collected into a sequence in order of evaluation. The qualified name for the referent Feature is resolved using the `result` parameter of the primary Expression as the context Namespace (see 7.2.4.2.4), but considering only public memberships.

```
// The primary Expression is "getPlatform(id)".
// The feature chain is "sensors.isActive".
// Results in a sequence of Boolean values,
// one for each platform sensor.
getPlatform(id).sensors.isActive
```

  To avoid ambiguity, the primary Expression of a feature chain expression shall not be itself a feature chain expression. To read a list of Features sequentially, rather than in a single evaluation, delimit nested feature chain expressions using parentheses

```
// First evaluate "getPlaform(id).sensors",
// then evaluate ".isActive" on the result of
// that.
(getPlatform(id).sensors).isActive
```

- *Collect expression.* A collect expression consists of a primary Expression and an Expression body (see 7.4.8.2.3) separated by a dot (`.`) symbol. The Expression body must have a single input parameter. The Expression body is evaluated on each of the result values from the primary Expression, in order, and each of the results are collected into a sequence in order of evaluation (that is, a collect expression is a shorthand for invoking the *`ControlFunctions::collect`* Function).

```
sensors.{in s: Sensor; s.reading}  // results in a sequence of
                                   // readings of each of the sensors
```

- *Select expression.* A select expression consists of a primary Expression and an Expression body (see 7.4.8.2.3) separated by a dot-question-mark (`.?`) symbol. The Expression body must have a single input parameter and a *Boolean* result. The Expression body is evaluated on each of the result values from the primary Expression, in order, and those for which the Expression body evaluates to true are selected for inclusion in the result of the select expression (that is, a select expression is a shorthand for invoking the *`ControlFunctions::select`* Function).

```
sensors.?{in s: Sensor; s.isActive}  // results in the subsequence of
                                     // sensors that are active
```

- *Function operation expression.* A function operation expression is a special syntax for an InvocationExpression in which the first argument is given before the arrow (`->`) symbol, which is followed by the name of the Function to be invoked and an argument list for any remaining arguments (see 8.21). This is useful for chaining invocations in an effective data flow.

```
sensors -> selectSensorsOver(limit) -> computeCriticalValue()
```

  If the invoked Function has exactly two input parameters, and the second input parameter is an Expression, then an Expression body (see 7.4.8.2.3) can be used as the argument for the second argument without surrounding parentheses. The argument Expression body should declare parameters consistent with those on the parameter Expression (if any). This is particularly useful when invoking Functions from the *ControlFunctions* library model (see 8.36).

```
sensors -> select {in s: Sensor; s::isActive}
members -> reject {in member: Member; !member->isInGoodStanding()}
factors -> reduce {in x: Real; in y: Real; x * y}
```

If the argument Expression is simply the direct invocation of another function, then the argument InvocationExpression may be specified using simply name of the invoked Function.

```
factors -> reduce RealFunctions::'*'
```

Table 8. Primary Expression Operator Mapping

Operator	Library Function	Description	Model-level Evaluable?
`[`	`BaseFunctions::'['`	Indexing	Yes
`,`	`BaseFunctions::','`	Sequence construction	Yes
`.`	`ControlFunctions::'.'`	Feature chaining	Yes
`collect`	`ControlFunctions::collect`	Sequence collection	No
`select`	`ControlFunctions::select`	Sequence selection	No

7.4.8.2.3 Base Expressions

```
BaseExpression : Expression =
      NullExpression
    | LiteralExpression
    | FeatureReferenceExpression
    | InvocationExpression
    | BodyExpression

NullExpression : NullExpression =
    'null' | '(' ')'

FeatureReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FeatureReferenceMember

FeatureReferenceMember : Membership =
    memberElement = FeatureReference

FeatureReference : Feature =
    [QualifiedName]

InvocationExpression : InvocationExpression =
    ownedRelationship += OwnedSpecialization
    ArgumentList(this)

ArgumentList (e : InvocationExpression) =
    '(' ( PositionalArgumentList(e) | NamedArgumentList(e) )? ')'

PositionalArgumentList (e : InvocationExpression) =
    e.ownedRelationship += ArgumentMember
    ( ',' e.ownedRelationship += ArgumentMember )*

ArgumentMember : ParameterMembership =
    ownedMemberParameter = Argument

Argument : Feature =
    ownedRelationship += ArgumentValue

NamedArgumentList (e : InvocationExpression) =
    e.ownedRelationship += NamedArgumentMember
    ( ',' e.ownedRelationship += NamedArgumentMember )*

NamedArgumentMember : FeatureMembership =
    ownedMemberFeature = NamedArgument

NamedArgument : Feature =
    ownedRelationship += ParameterRedefinition '='
    ownedRelationship += ArgumentValue

ParameterRedefinition : Redefinition =
    redefinedFeature = [QualifiedName]

ArgumentValue : FeatureValue =
    value = OwnedExpression

BodyExpression : FeatureReferenceExpression =
    ownedRelationship += ExpressionBodyMember

ExpressionBodyMember : FeatureMembership =
```

```
    ownedMemberFeature = ExpressionBody

ExpressionBody : Expression =
    FunctionBody(this)
```

The *base expressions* include representations for NullExpressions, LiteralExpressions, InvocationExpressions and FeatureReferenceExpressions.

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 7.4.8.2.4.

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 InvocationExpression is considered to subset the named Expression, meaning that, effectively, 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.
}
```

It is also possible to specify an Expression to be invoked using a Feature chain (see 7.3.4.2.6).

```
class Stats {
    feature vales[1..*] : Real;
    expr avg { sum(values)/size(values) }
}
feature myStats : States {
    redefines feature values = (1.0, 2.0, 3.0);
}
feature myAvg = myStats.avg();
```

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 having following parentheses.

```
expr addOne : UnaryFunction(x : Anything): Integer {
    x istype Integer? (x as Integer) + 1: 0
}
feature two = apply(addOne, 1); // "addOne" is a reference to expr addOne
```

Rather than declaring a named Expression in order to pass it as an argument, an Expression body may be used directly as a base expression. In this case, any parameters must be declared as Features with direction within the Expression body (see 7.4.6.2.1). Such body expressions are particularly useful when used for the second argument of a function operation expression (see 7.4.8.2.2).

```
feature two = apply({in x; x istype Integer? (x as Integer) + 1: 0}, 1);
feature incrementedValues = values -> collect {in x: Number; x + 1};
```

7.4.8.2.4 Literal Expressions

```
LiteralExpression : LiteralExpression =
      LiteralBoolean
    | LiteralString
    | LiteralInteger
    | LiteralReal
    | LiteralInfinity

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

LiteralInfinity : LiteralInfinity =
    '*'
```

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 7.4.8.2.1) 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 LiteralInfinity is represented by the symbol `*`.

7.4.8.3 Abstract Syntax

7.4.8.3.1 Overview

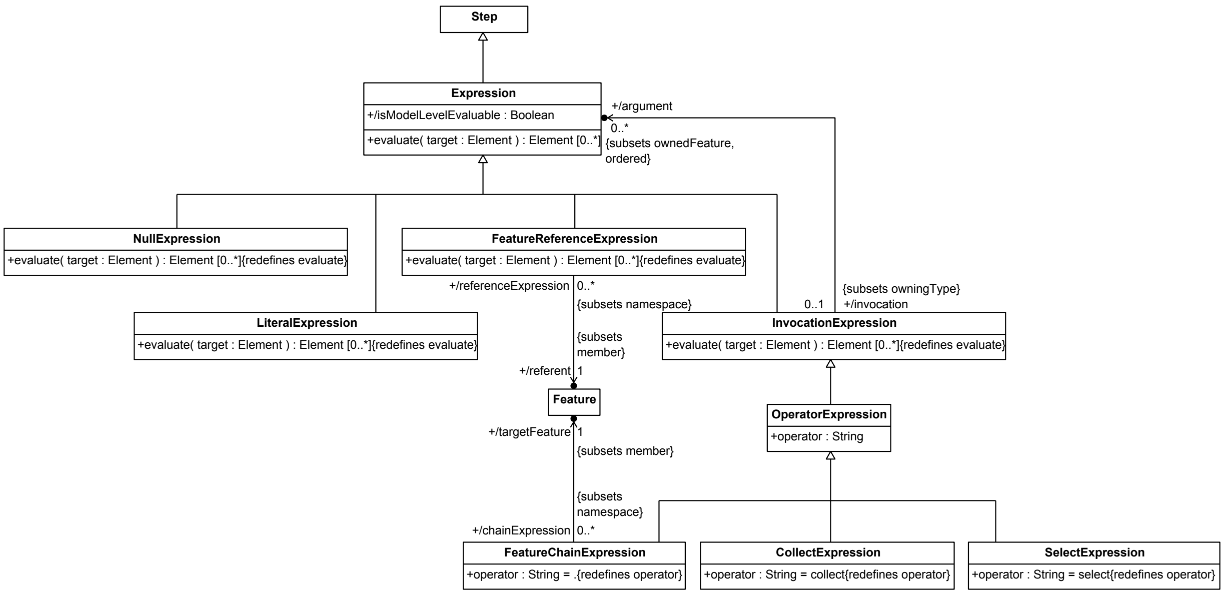

Figure 28. Expressions

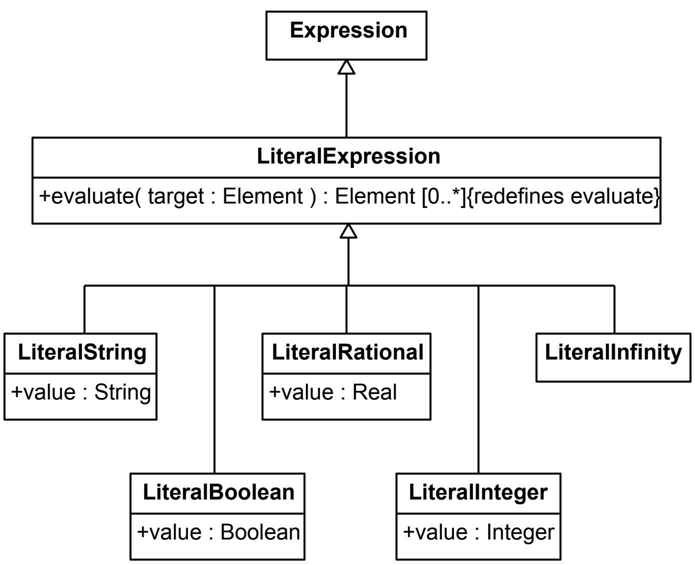

Figure 29. Literal Expressions

7.4.8.3.2 CollectExpression

Description

A CollectExpression is an OperatorExpression whoise operator is `"collect"`, which resolves to the library Function *`ControlFunctions::collect`*.

General Classes

OperatorExpression

Attributes

operator : String {redefines operator}

Operations

No operations.

Constraints

None.

7.4.8.3.3 FeatureChainExpression

Description

A FeatureChainExpression is an OperatorExpression whose operator is `"."`, which resolves to the library Function *`ControlFunctions::'.'`*. It evaluates to the result of chaining the `result` Feature of its single `argument` Expression with its `targetFeature`.

The first two `members` of a FeatureChainExpression must be its single `argument` Expression and its `targetFeature`. Its only other `members` shall be those necessary to complete it as an InvocationExpression.

General Classes

OperatorExpression

Attributes

operator : String {redefines operator}

/targetFeature : Feature {subsets member}

The Feature that is accessed by this FeatureChainExpression, derived as its second `member` Feature (the first being its one `argument` Expression). This Feature must redefine the *`target` Feature of the Function `ControlFunctions::'.'`*.

Operations

No operations.

Constraints

None.

7.4.8.3.4 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 {subsets member}

The Feature that is referenced by this FeatureReferenceExpression, derived as its first `member` Feature.

Operations

evaluate(target : Element) : Element [0..*]

If the `target` Element is a Type that has a `feature` that redefines the `referent`, then return the result of evaluating the Expression given by the FeatureValue of that `feature`. Otherwise, if the `referent` has no `featuringTypes`, return the `referent`. Otherwise return an empty sequence.

```
body: if not target.oclIsKindOf(Type) then Sequence{}
else
    let feature: Sequence(Feature) =
        target.oclAsType(Type).feature->select(f |
            f.ownedRedefinition.redefinedFeature->
                includes(referent)) in
        if feature->notEmpty() then
            feature.valuation.value.evaluate(target)
        else if referent.featuringType->isEmpty()
            then referent
        else Sequence{}
        endif endif
endif
```

Constraints

featureReferenceExpressionIsModelLevelEvaluable

A FeatureReferenceExpression is always model-level evaluable (though it may produce no value on some targets).

7.4.8.3.5 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

evaluate(target : Element) : Element [0..*]

Apply the Function that is the `type` of this InvocationExpression to the argument values resulting from evaluating each of the `argument` Expressions on the given `target`. If the application is not possible, then return an empty sequence.

Constraints

invocationExpressionIsModelLevelEvaluable

An InvocationExpression is model-level evaluable if all its `argument` Expressions are model-level evaluable and its `function` is model-level evaluable.

```
isModelLevelEvaluable =
    argument->forAll(isModelLevelEvaluable) and
    function.isModelLevelEvaluable
```

7.4.8.3.6 LiteralBoolean

Description

LiteralBoolean is a LiteralExpression that provides a *Boolean* value as a result. It 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

None.

7.4.8.3.7 LiteralExpression

Description

A LiteralExpression is an Expression that provides a basic value as a result. It 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

None.

Operations

evaluate(target : Element) : Element [0..*]

The model-level value of a LiteralExpression is itself.

```
body: Sequence{self}
```

Constraints

literalExpressionIsModelLevelEvaluable

A LiteralExpression is always model-level evaluable.

```
isModelLevelEvaluable = true
```

7.4.8.3.8 LiteralInteger

Description

A LiteralInteger is a LiteralExpression that provides an Integer value as a result. It 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

None.

7.4.8.3.9 LiteralReal

Description

A LiteralRational is a LiteralExpression that provides a Rational value as a result. It must have an owned `result` parameter whose type is *Rational*.

General Classes

LiteralExpression

Attributes

value : Real

The value whose rational approximation is the result of evaluating this Expression.

Operations

No operations.

Constraints

None.

7.4.8.3.10 LiteralString

Description

A LiteralString is a LiteralExpression that provides a String value as a result. It 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

None.

7.4.8.3.11 LiteralInfinity

Description

A LiteralInfinity is a LiteralExpression that provides the positive infinity value ("*"). It must have an owned `result` parameter whose type is *Positive*.

General Classes

LiteralExpression

Attributes

None.

Operations

No operations.

Constraints

None.

7.4.8.3.12 NullExpression

Description

A NullExpression is an Expression that results in a null value. It must be typed by a *NullEvaluation* that results in an empty value.

General Classes

Expression

Attributes

None.

Operations

evaluate(target : Element) : Element [0..*]

The model-level value of a NullExpression is an empty sequence.

```
body: Sequence{}
```

Constraints

nullExpressionIsModelLevelEvaluable

A NullExpression is always model-level evaluable.

```
isModelLevelEvaluable = true
```

7.4.8.3.13 OperatorExpression

Description

An OperatorExpression is an InvocationExpression whose `function` is determined by resolving its `operator` in the context of one of the standard Function packages from the Kernel Model Library.

General Classes

InvocationExpression

Attributes

/operand : Expression [0..*] {ordered}

operator : String

An operator symbol that names a corresponding Function from one of the standard Function packages from the Kernel Model Library .

Operations

No operations.

Constraints

None.

7.4.8.3.14 SelectExpression

Description

A CollectExpression is an OperatorExpression whoise operator is `"select"`, which resolves to the library Function *`ControlFunctions::select`*.

General Classes

OperatorExpression

Attributes

operator : String {redefines operator}

Operations

No operations.

Constraints

None.

7.4.8.4 Semantics

Required Specializations of Model Library

1. LiteralExpressions shall directly or indirectly specialize *`Performances::literalEvaluations`* (see 8.6), which means their `function` is *Performances::LiteralEvaluation*s or a specialization of it.
2. NullExpressions shall directly or indirectly specialize *`Performances::nullEvaluations`* (see 8.6), which means their `function` is *Performances::NullEvaluations* or a specialization of it.

Also see Required Generalizations for Expressions in 7.4.7.4.

Null Expression Semantics

Invocations of NullExpressions do not produce any `result` values (see rules above and 7.4.8.1).

Literal Expression Semantics

With the exception of LiteralInfinity, each kind of LiteralExpression has a `value` meta-property with its own primitive Type, which is given a required constant value in models to specify the value of the *`result`* of *LiteralEvaluations* classified by each LiteralExpression (see 8.6.1). LiteralInfinity does not have a `value` property, because its `result` parameter value is always a number greater than all the integers ("infinity"), but treated like one, notated by `*`, from the standard DataType *Natural*.

LiteralExpressions are Expressions that have the values of their `result` parameter specified as a constant in models by a LiteralExpression's `value` property, ultimately being *DataValues* in the `result` parameter of *LiteralEvaluations* classified by the LiteralExpression (see 8.6.1). LiteralInfinities are LiteralExpressions resulting in a number greater than all the integers ("infinity"), but treated like one, notated as `*` (see 8.18.1).

Release Note. The semantics of literals will be more formally addressed in the final submission.

Feature Reference Expression Semantics

A FeatureReferenceExpression for a Feature `f` is semantically equivalent the Expression

```
expr () result {
    binding result = f;
}
```

where the types of the `result` parameter are considered to be implicitly the same as those of `f`.

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 7.4.8.2.1) 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

```
DataFunctions::'!' (expr)
```

and a binary operator Expression such as

```
expr_1 + expr_2
```

is equivalent to the InvocationExpression

```
DataFunctions::'+' (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.

Release 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 final submission.

Expression Body Semantics

An Expression body used as a base expression (see 7.4.8.2.3) is equivalent to a FeatureReferenceExpression that contains the Expression body as its own referent. That is, a Expression body of the form

```
{ body }
```

is semantically equivalent to

```
expr () result {
    expr e () result { body }
    binding result = e;
}
```

However, when an Expression body is used as the argument to an invocation, this can be more directly realized by directly binding to the Expression body without the intermediate FeatureReferenceExpression. Thus, the invocation

```
F({ body })
```

is semantically equivalent to

```
expr e : F (a) result {
    expr e_1 () result { body }
    binding a = e_1;
}
```

Note that the binding is to `e_1` itself, *not* `e_1.result`.

Control Function Invocation Semantics

Certain operator expressions (see 7.4.8.2.1) denote invocations of Functions in the *ControlFunctions* library model that have one or more parameters that are Expressions (see 8.36). The arguments corresponding to these parameters are handled by special rules that wrap the given argument Expressions in Expression bodies so they can be passed without being immediately evaluated.

The second and third operands of the ternary conditional test operator `?` are for Expression parameters. Therefore, a conditional test Expression of the form

```
expr_1 ? expr_2 : expr_3
```

is semantically equivalent to

```
ControlFunctions::'?'(expr_1, { expr_2 }, { expr_3 })
```

The the second operand of the binary conditional logical operators `&&` and `||` is for an Expression parameter. Therefore, a conditional logical Expression of the form

```
expr_1 && expr_2
```

is semantically equivalent to

```
ControlFunctions::'&&'(expr_1, { expr_2 })
```

and similarly for `||`.

Model-Level Evaluable Expression Semantics

As defined in 7.4.7.2.2, a model-level evaluable Expression is an Expression that can be evaluated using metadata available within a model itself. This means that the evaluation rules for such an expression can be defined entirely within the abstract syntax. If an Expression is model-level evaluable, then using `evaluate` operation on it gives the model-level evaluation of the Expression as an ordered list of Elements.

A model-level evaluable Expression is evaluated on a given *target* object (see 7.4.12.4 and 7.4.13.4 for the targets used in the case of metadata `values` and `filterConditions`, respectively), according to the following rules.

1. A NullExpression evaluates to the empty list.
2. A LiteralExpression evaluates to itself.
3. An FeatureReferenceExpression evaluates to one of the following.
    - If the target Element has a MetadataFeature (see 7.4.12) with a nested Feature that redefines the `referent`, then the FeatureReferenceExpression evaluates to the result of evaluating the corresponding bound `value` expression on the same target Element (if any).
    - Otherwise, if the `referent` is a Feature with no FeaturingTypes or with *Anything* as a FeaturingType, then the FeatureReferenceExpression evaluates to the `referent`.
    - Otherwise, the FeatureReferenceExpression evaluates to the empty list.
4. An InvocationExpression evaluates to an application of its `function` to argument values corresponding by the results of evaluating each of the `argument` Expressions of the InvocationExpression, with the correspondence as given below.

Every Element in the list resulting from a model-level evaluation of an Expression according to the above rules will be either a LiteralExpression or a Feature of *Anything.* If each of these Elements is further evaluated according to its regular instance-level semantics, then the resulting list of instances will correspond to the result that would be obtained by evaluating the original Expression using its regular semantics on the referenced metadata of the target Element.

Release Note. In the final submission, the semantics of model-level evaluation may be more formally defined as Expression evaluation on a reflective KerML abstract syntax model of the KerML.

7.4.9 Interactions

7.4.9.1 Interactions Overview

Interactions

Interactions are Behaviors that are also Associations (see 7.4.6 and 7.4.4, respectively), classifying *Performances* that are also *Links* between *Occurrences* (see 8.3 through 8.6). They specify how (*Link*) participants affect each other and collaborate.

Transfers are Interactions between two participants (binary Interactions, see 8.7). They specify when things provided by one *Occurrence* (via its output Features) are accepted by another (via input Features).

Item Flows

ItemFlows are Steps that are also binary Connectors (see 7.4.6 and 7.4.5, respectively) typed only by *Transfer* (see 8.7), or its specializations. ItemFlow's values (*Transfers*) ensure the outputs of values of one connected Feature (`sourceFeature`) will be the same as inputs of another (`targetFeature`), where outputs and inputs are values of the `sourceOutputFeature` and `targetInputFeature`, respectively, which must be classified by `itemType`.

SuccessionItemFlows are ItemFlows that are also Successions (see 7.4.5), typed by *TransferBefore* (see 8.7). They identify (have as values) *TransferBefores* that happen after their *`source`* (the end of an *Occurrence* that provides the things being transferred) and before their *`target`* (the start of an *Occurrence* accepting those things).

7.4.9.2 Concrete Syntax

7.4.9.2.1 Interactions

```
Interaction : Interaction =
    ( isAbstract ?= 'abstract' )? 'interaction'
    BehaviorDeclaration(this) TypeBody(this)
```

An Interaction is declared as a Behavior (see 7.4.6.2.1), 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 8.6) and the Association *BinaryLink* or the Class *Link* from the *Objects* library model (see 8.5), 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 7.4.6.2.1). 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 Association that is a `subclassifier` (see 7.4.4.2).

```
interaction Authorization {
    end feature client[*] : Computer;
    end feature server[*] : Computer;
    composite step login;
    composite step authorize;
    composite succession login then authorize;
}
```

7.4.9.2.2 Item Flows

```
ItemFlow (m : Membership) : ItemFlow =
    FeaturePrefix 'flow'
    ItemFlowDeclaration(this, m) TypeBody(this)

SuccessionItemFlow (m : Membership) : SuccessionItemFlow =
    FeaturePrefix 'succession' 'flow'
    ItemFlowDeclaration(this, m) TypeBody(this)

ItemFlowDeclaration (i : ItemFlow, m : Membership) =
     ( FeatureDeclaration(i, m)
       ( 'of'  i.ownedRelationship += ItemFeatureMember
       | i.ownedRelationship += EmptyItemFeatureMember )
       'from'
     | ( isSufficient ?= 'all' )?
       i.ownedRelationship += EmptyItemFeatureMember
     )
    i.ownedRelationship += ItemFlowEndMember 'to'
    i.ownedRelationship += ItemFlowEndMember

ItemFeatureMember : FeatureMembership =
    ownedMemberFeature = ItemFeature

ItemFeature : Feature =
     ( memberName = NAME ':' )?
     ( ownedTyping += OwnedFeatureTyping
       ( ownedRelationship += OwnedMultiplicity )?
     | ownedRelationship += OwnedMultiplicity
       ( ownedTyping += OwnedFeatureTyping )?
     )

EmptyItemFeatureMember : FeatureMembership =
    ownedMemberFeature = EmptyItemFeature

EmptyItemFeature : Feature =
    {}

ItemFlowEndMember : FeatureMembership =
    ownedMemberFeature = ItemFlowEnd

ItemFlowEnd : Feature =
     ( ownedRelationship += Subsetting '.' )?
    ownedRelationship += ItemFlowFeatureMember

ItemFlowFeatureMember : FeatureMembership =
    ownedMemberFeature = ItemFlowFeature

ItemFlowFeature : Feature =
      ownedRelationship += ItemFlowRedefinition

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

An ItemFlow declaration is syntactically similar to a binary Connector declaration (see 7.4.5.2.1), using the keyword **`flow`**, or **`succession 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 7.4.5.2.1) 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.
    flow fuelFlow from fuelTank::fuelOut to engine::fuelIn;
}
```

The `sourceOutput` and `targetInput` of an ItemFlow can also be specified using Feature paths (see 7.4.5.2.1). In this case, the `relatedFeatures` are determined as the featured identified by the paths, excluding the last feature. This is particularly useful when the desired `relatedFeatures` are inherited Features.

```
class Vehicle {
    composite feature fuelTank {
        out feature fuelOut : Fuel;
    }
    composite feature engine {
        in feature fuelIn : Fuel;
    }
}

feature vehicle : Vehicle {
    // The ItemFlow actually connects the inherited fuelTank
    // Feature to the inherited engine Feature.
    flow 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.

```
flow 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.

```
flow 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.
```

```
    succession 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 8.7), or to the SuccessItemFlow *`transfersBefore`*, 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 7.4.6.2.2).

7.4.9.3 Abstract Syntax

7.4.9.3.1 Overview

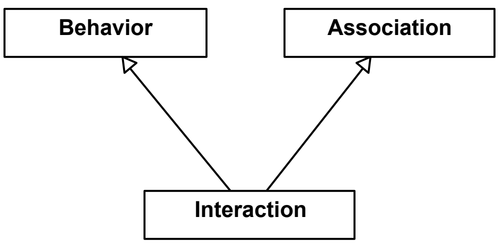

Figure 30. Interactions

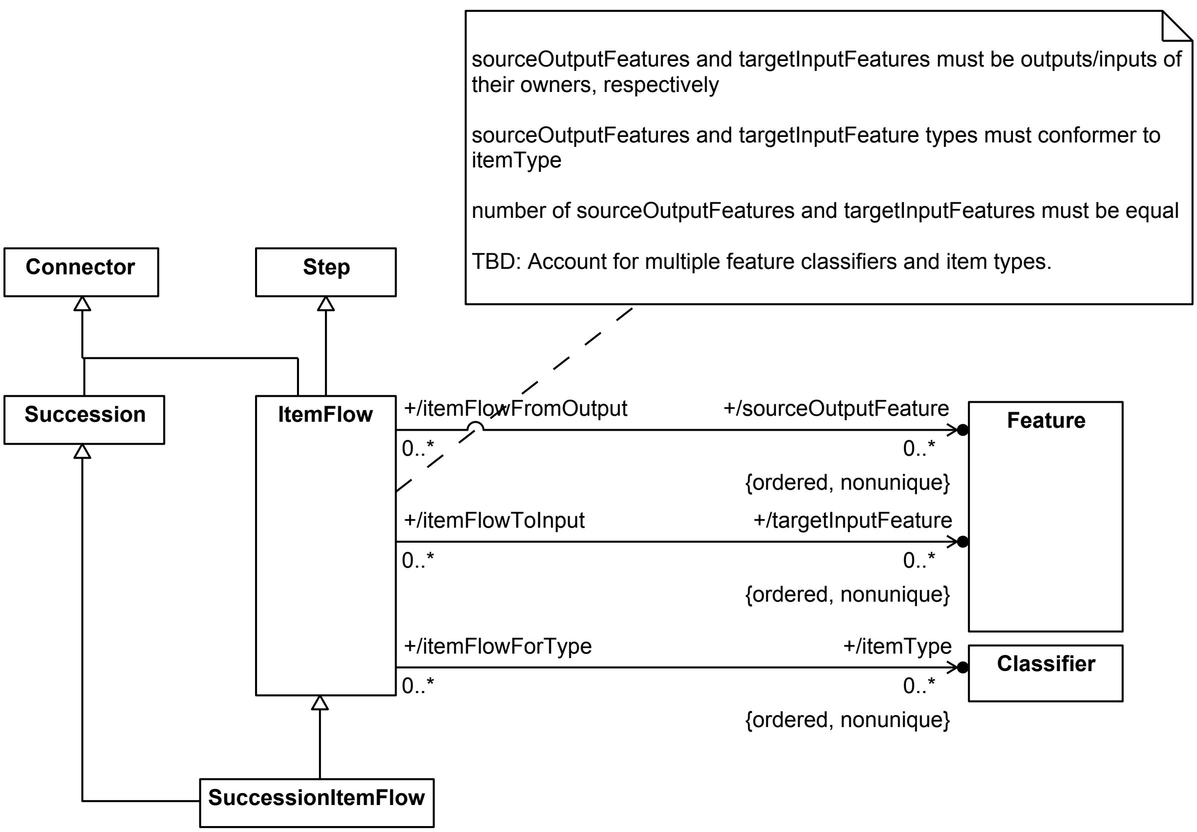

Figure 31. Item Flows

The Features that originate the ItemFlow. They must be owned `outputs` of the source participant of the ItemFlow. If there are no such Features, then the ItemFlow must be abstract.

7.4.9.3.2 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.

An ItemFlow must be typed by the Interaction *Transfer* from the Kernel library, or a specialization of it.

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 [0..*] {ordered, nonunique}

The type of the item transferred, derived as the `type` of the `feature` of the ItemFlow that directly or indirectly redefines *`Transfer::item`*.

/sourceOutputFeature : Feature [0..*] {ordered, nonunique}

The Feature that originates the ItemFlow.

/targetInputFeature : Feature [0..*] {ordered, nonunique}

The Features that receive the ItemFlow. They must be owned `outputs` of the target participant of the ItemFlow. If there are no such Features, then the ItemFlow must be abstract.

Operations

No operations.

Constraints

None.

7.4.9.3.3 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

None.

Operations

No operations.

Constraints

None.

7.4.9.3.4 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.

A SuccessionItemFlow must be typed by the Interaction *TransferBefore* from the Kernel Library, or a specialization of it.

General Classes

Succession
ItemFlow

Attributes

None.

Operations

No operations.

Constraints

None.

7.4.9.4 Semantics

Required Specializations of Model Library

1. Interactions shall directly or indirectly specialize *Link::Link* (see 8.3.2.3), or *Links::BinaryLink* (see 8.3.2.1) for Interactions with exactly two participants.
2. Interactions shall directly or indirectly specialize *Performances::Performance* (see 8.6.2.11).
3. ItemFlows shall directly or indirectly specialize *`Transfers::transfers`* (see 8.7.2.3), which means they shall be typed by (a specialization of) *Transfers::Transfer* (see 8.7.2.1).
4. The `connectorEnds` of ItemFlows shall
    a. Redefine *`source`* and *`target`* of *Transfers::Transfer* (see 8.7.2.1).
    b. Nest Features that redefine *`source::sourceOutput`* and *`target::targetInput`*, and subset the `sourceOutputFeature` and `targetInputFeature` of the ItemFlow.
5. ItemFlows that specify the kind of item flowing (`itemType`) shall add an `ownedFeature` that directly or indirectly redefines *`Transfer::item`* with that Type.
6. SuccessionItemFlows directly or indirectly specialize *`Transfers::flows`* (see 8.7.2.4), which means they shall be typed by (a specialization of) *Transfers::TransferBefore* (see 8.7.2.2).

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

```
classifier I specializes Link::BinaryLink, Performances::Performance {
    end feature e1 redefines Link::BinaryLink::source {
        feature e2 = I::e2(e1);
    }
    end feature e2 redefines Link::BinaryLink::target {
        feature e1 = I::e1(e2);
    }
    in feature x;
    out feature y;
    inout feature z;
}
```

Item Flow Semantics

An ItemFlow of the form

```
flow of item : T from f1.f1_out to f2.f2_in;
```

is semantically equivalent to the core model

```
feature subsets Transfers::transfers {
    end feature redefines source subsets f1 {
        feature redefines sourceOutput subsets f1_out;
    }
    end feature redefines target subsets f2 {
        feature redefines targetInput subsets f2_in;
    }
}
```

A SuccessionItemFlow is semantically the same, except that *`Transfers::transfersBefore`* is used instead of *`transfers`*.

7.4.10 Feature Values

7.4.10.1 Feature Values Overview

A FeatureValue is a Membership that identifies a particular member Expression (`value`) whose `result` provides the value of the Feature that owns the FeatureValue. The value is specified as either a bound value (`isInitial = false`) or an initial value (`isInitial = true`), and as either a concrete value (`isDefault = false`) or default value (`isDefault = true`). A Feature shall have at most one FeatureValue.

7.4.10.2 Concrete Syntax

```
ValuePart (f : Feature) =
    f.ownedRelationship += FeatureValue

FeatureValue : FeatureValue =
    ( '='
    | isInitial ?= ':='
    | isDefault ?= 'default' ( '=' | isInitial ?= ':=' )?
    )
    value = OwnedExpression
```

A FeatureValue with `isDefault = false` and `isInitial = false` is declared using the symbol = followed by a representation of the `value` Expression using the concrete syntax from 7.4.8.2. This notation is appended to the declaration of the Feature that is the `featureWithValue` for the FeatureValue.

```
feature monthsInYear : Natural = 12;
struct TestRecord {
    feature scores[1..*] : Integer;
    derived feature averageScore[1] : Rational = sum(scores)/size(scores);
}
```

Features that have a FeatureValue of this form shall also have a nested BindingConnector between the Feature and `result` of the `value` Expression.

Note. The semantics of binding mean that such a FeatureValue asserts that a Feature is *equivalent* to the result of the value Expression (see 7.4.5.4 on the semantics of BindingConnectors). To highlight this, a Feature with such 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 – see also 7.3.4.2.1).

A FeatureValue with `isDefault = false` and `isInitial = true` is declared as above but using the symbol `:=` instead of =.

```
feature count : Natural := 0;
```

In this case, the Feature shall also have a nested Step typed by a FeatureWritePerformance (see 8.8.2.8) used to initialize the Feature to the result of the value Expression. Unlike in the case of a bound value, an initial value may be changed using subsequent FeatureWritePerformances.

A FeatureValue with `isDefault = true` and either value for `isInitial` is declared similarly to the above, but with the keyword **`default`** preceding the symbol `=` or `:=`. However, for `isInitial = false`, the symbol `=` may be elided.

```
struct Vehicle {
    feature mass : Real default 1500.0;
    feature engine : Engine default := standardEngine;
}
struct TestWithCutoff :> TestRecord {
    feature cutoff : Rational default = 0.75 * averageScore;
}
```

If `isDefault = true`, then no BindingConnector or intialization Step shall be added to the Feature.

A FeatureValue can be included with the following kinds of Feature declaration:

- Feature (see 7.3.4.2.1)
- Step (see 7.4.6.2.2)
- Expression (see 7.4.7.2.2)
- BooleanExpression and Invariant (see 7.4.7.2.4)

A FeatureValue can also be used in the declaration of a `parameter` in a Step or Expression declaration (see 7.4.6.2.2 and 7.4.7.2.2).

```
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);
}
```

7.4.10.3 Abstract Syntax

7.4.10.3.1 Overview

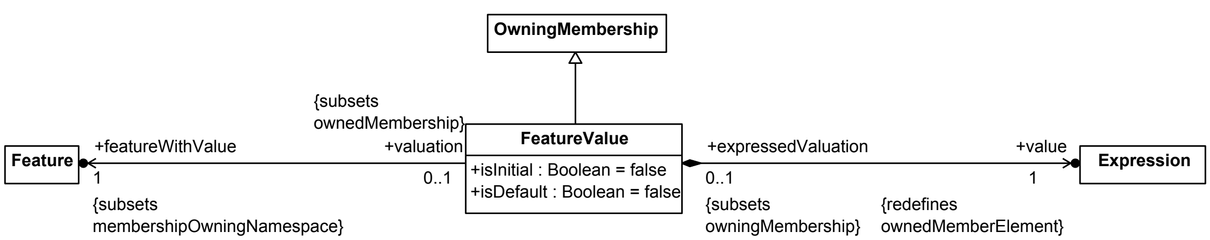

Figure 32. Feature Values

7.4.10.3.2 FeatureValue

Description

A FeatureValue is a Membership that identifies a particular member Expression that provides the value of the Feature that owns the FeatureValue. The value is specified as either a bound value or an initial value, and as either a concrete or default value. A Feature can have at most one FeatureValue.

If `isInitial = false`, then the result of the `value` expression is bound to the `featureWithValue` using a BindingConnector. Otherwise, the `featureWithValue` is initialized using a FeatureWritePeformance.

If `isDefault = false`, then the above semantics of the FeatureValue are realized for the given `featureWithValue`. Otherwise, the semantics are realized for any individual of the `featuringType` of the `featureWithValue`, unless another value is explicitly given for the `featureWithValue` for that individual.

General Classes

OwningMembership

Attributes

featureWithValue : Feature {subsets membershipOwningNamespace}

The Feature to be provided a value.

isDefault : Boolean

Whether this FeatureValue is a concrete specification of the bound of initial value of the `featureWithValue`, or just a default value that may be overridden.

isInitial : Boolean

Whether this FeatureValue specifies a bound value or an initial value for the `featureWithValue`.

value : Expression {redefines ownedMemberElement}

The Expression that provides the value of the `featureWithValue` as its `result`.

Operations

No operations.

Constraints

featureValueBindingConnector

The `valueConnector` must be an `ownedMember` of the `featureWithValue` whose `relatedElements` are the `featureWithValue` and the `result` of the `value` Expression and whose `featuringTypes` are the same as those of the `featureWithValue`.

```
valueConnector.owningNamespace = featureWithValue and
valueConnector.relatedFeature->includes(featureWithValue) and
valueConnector.relatedFeature->includes(value.result) and
valueConnector.featuringType = featureWithValue.featuringType
```

featureValueExpressionDomain

The `value` Expression must have the same `featuringTypes` as the `featureWithValue`.

```
value.featuringType = featureWithValue.featuringType
```

7.4.10.4 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 7.4.8.4.

7.4.11 Multiplicities

7.4.11.1 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 Multiplicities in 7.3.2.1). 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, with both Expression `result` parameters typed by *Natural* (see 8.18). An `upperBound` value of * (infinity) means that the cardinality includes all numbers greater than or equal to the `lowerBound` value.

Release Note. Supporting additional kinds of Multiplicities (such as multiple ranges like `[2..4, 6..8]`) will be considered for the final submission.

7.4.11.2 Concrete Syntax

```
Multiplicity : Multiplicity =
    MultiplicitySubset | MultiplicityRange

MultiplicitySubset : Multiplicity =
    'multiplicity' Identification(this) Subsets(this) ';'

MultiplicityRange : MultiplicityRange =
    'multiplicity' Identification(this) MultiplicityBounds ';'

OwnedMultiplicity : OwningMembership =
    ownedMemberElement = OwnedMultiplicityRange

OwnedMultiplicityRange : MultiplicityRange =
    MultiplicityBounds

MultiplicityBounds : MultiplicityRange =
    '[' ( ownedRelationship += MultiplicityExpressionMember '..' )?
          ownedRelationship += MultiplicityExpressionMember ']'

MultiplicityExpressionMember : OwningMembership =
    ownedMemberElement = ( LiteralExpression | FeatureReferenceExpression )
```

A MultiplicityRange is written in the form `[`*`lowerBound`*`..`*`upperBound`*`]`, where each of *`lowerBound`* and *`upperBound`* is either a LiteralExpression or a FeatureReferenceExpression represented in the notation described in

7.4.8. LiteralExpressions can be used to specify a MultiplicityRange with fixed lower and/or upper bounds. The `type` of the `result` parameter of these Expressions shall be *Natural* (see 8.18) or a direct or indirect specialization of it.

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 infinite value `*`, in which case the lower bound is taken to be 0. If two Expressions are given, and the result of the first Expression is `*`, then the meaning of the MultiplicityRange is not defined.

```
class Automobile {
    feature n : Positive;
    composite feature wheels : Wheel[n]; // Equivalent to [n..n]
    feature driveWheels[2..n] subsets wheels;
}
feature autoCollection : Automobile[*]; // Equivalent to [0..*]
```

A named Multiplicity is declared using the keyword **`multiplicity`** followed by a `shortName` and/or `name` and the Multiplicity bounds. A Multiplicity can also be declared to be a subset of another Multiplicity.

```
multiplicity zeroOrMore [0..*];
multiplicity m subsets zeroOrMore;
```

If a named Multiplicity is declared in the body of a Feature, then then this shall be the multiplicity of the Feature. A Feature shall have at most one multiplicity, whether this is given in the declaration or the body of the Feature.

```
feature driveWheels subsets wheels {
    multiplicity [2..n];
}
feature autoCollection {
    multiplicity subsets zeroOrMore;
}
```

7.4.11.3 Abstract Syntax

7.4.11.3.1 Overview

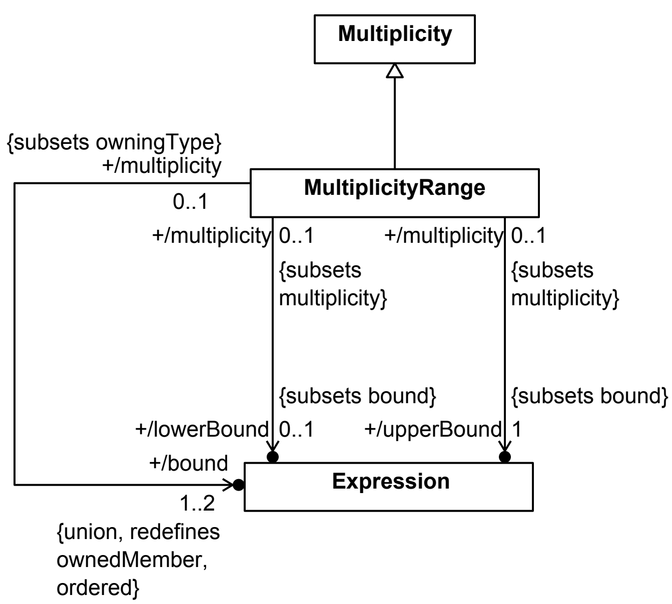

Figure 33. Multiplicities

7.4.11.3.2 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 ownedMember, ordered, union}

The bound Expressions of the MultiplicityRange. These shall be the only `ownedMembers` 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

multiplicityRangeExpressionDomain

The `bounds` of a MultiplicityRange shall have the same `featuringTypes` as the MultiplicityRange.

```
bound->forAll(b | b.featuringType = self.featuringType)
```

7.4.11.4 Semantics

Required Specializations of Model Library

1. MultiplicityRanges shall directly subset *`Base::naturals`* (see 8.2.2.4), 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 8.18.2.4) 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.

7.4.12 Metadata

7.4.12.1 Metadata Overview

Metadata is additional information on Elements of a model that does not have any instance-level semantics (in the sense described in 7.3.1.1) given in this specification. In general, metadata is specified in AnnotatingElements (including Comments and TextualRepresentations) attached to `annotatedElements` (see 7.2.3). A MetadataFeature is a kind of AnnotatingElement that allows for the definition of structured metadata with modeler-specified attributes. This may be used, for example, to add tool-specific information to a model that can be relevant to the function various kinds of tooling that may use or process a model, or domain-specific information relevant to a certain project or organization.

An MetadataFeature is syntactically a Feature (see 7.3.4) that is typed by a single Metaclass, which is a kind of Structure (see 7.4.3), with implicit multiplicity `1..1`. If the Metaclass has no `features`, then the MetadataFeature simply acts as a user-defined syntactic tag on the `annotatedElement`. If the Metaclass has `features`, then the MetadataFeature must have nested `features` that redefine each of the `features` of its `type`, binding them to the results of model-level evaluable Expressions (see 7.4.8), which provide the values of the specified attributive metadata for the `annotatedElement`.

7.4.12.2 Concrete Syntax

```
Metaclass : Metaclass =
    ( isAbstract ?= 'abstract' )? 'metaclass'
    ClassifierDeclaration(this) TypeBody(this)

OwnedMetadataFeatureAnnotation : Annotation =
    ownedRelatedElement += OwnedMetadataFeature

OwnedMetadataFeature : MetadataFeature
    ( '@' | 'metadata' )
    ( Identification(this) ( ':' | 'typed' 'by' ) )?
    ownedRelationship += MetadataTyping
    TypeBody

MetadataFeature : MetadataFeature =
    ( '@' | 'metadata' )
    MetadataFeatureDeclaration(this)
    ( 'about' annotation += Annotation
      { ownedRelationship += annotation }
      ( ',' annotation += Annotation
        { ownedRelationship += annotation } )*
    )?
    MetadataBody

MetadataFeatureDeclaration (f : MetadataFeature, m : Membership) =
    ( Identification(f, m) ( ':' | 'typed' 'by' ) )?
    f.ownedRelationship += ownedFeatureTyping

MetadataBody (f : MetadataFeature) =
    ';' | '{' ( f.ownedRelationship += MetadataBodyElement )* '}'

MetadataBodyElement : Membership =
      NonFeatureMember
    | MetadataBodyFeatureMember
    | AliasMember
    | Import

MetadataBodyFeatureMember : FeatureMembership =
    ownedMemberFeature = MetadataBodyFeature

MetadataBodyFeature : MetadataFeature =
    'feature'? ( ':>>' | 'redefines')? ownedRelationship += OwnedRedefinition
    FeatureSpecializationPart? ValuePart?
    MetadataBody
```

A Metaclass is declared like a Structure (see 7.4.3), but using the keyword **`metaclass`**. If no `ownedSuperclassing` is explicitly given for the Metaclass, then it is implicitly given a default Superclassing to the Metaclass *`Metaobject`* from the *`Metaobjects`* library model (see 8.16).

```
metaclass SecurityRelated;

metaclass ApprovalAnnotation {
    feature approved : Boolean;
    feature approver : String;
}
```

A MetadataFeature is declared using the keyword **metadata** (or the symbol `@`), optionally followed by a `shortName` and/or `name`, followed by the keyword **typed by** (or the symbol **:**) and the qualified name of exactly one Metaclass. If no `shortName` or `name` is given, then the keyword **typed by** (or the symbol **:**) may also be omitted. One or more `annotatedElements` are then identified for the MetadataFeature after the keyword **about**, indicating that the MetadataFeature has Annotation Relationships to each of the identified Elements (see 7.2.3).

```
metadata securityDesignAnnotation : SecurityRelated about SecurityDesign;
```

If the specified Metaclass has `features`, then a body must be given for the MetadataFeature that declares Features that redefine each of the `features` of the Metaclass and binds them to the result of model-level evaluable Expressions (see 7.4.7.3.3). The nested Features of a MetadataFeature must always have the same names as the names of the typing Metaclass, so the shorthand prefix redefines notation (see 7.3.4.2.5) is always used.

```
metadata ApprovalAnnotation about Design {
    feature redefines approved = true;
    feature redefines approver = "John Smith";
}
```

The keywords **feature** and/or **redefines** (or the equivalent symbol `:>>`) may be omitted in the declaration of a MetadataFeature.

```
metadata ApprovalAnnotation about Design {
    approved = true;
    approver = "John Smith";
}
```

If the MetadataFeature is an `ownedMember` of a Namespace (see 7.2.4), then the explicit identification of `annotatedElements` can be omitted, in which case the `annotatedElement` shall be implicitly the containing Namespace (see 7.2.3).

```
class Design {
    // This MetadataFeature is implicitly about the class Design.
    @ApprovalAnnotation {
        approved = true;
        approver = "John Smith";
    }
}
```

If a MetadataFeature has one or more concrete `features` that directly or indirectly subset *`Metaobject::annotatedElement`*, then, for each `annotatedElement` of the MetadataFeature, there shall be at least one such Feature for which the metaclass of the `annotatedElement` conforms to all the types of the Feature (which must all be specializations of the reflective Metaclass *KerML::Element,* see 8.17).

```
metaclass Command {
    // A MetadataFeature of this Metaclass may annotate
    // a Behavior or a Step.
    subsets annotatedElement : KerML::Behavior;
    subsets annotatedElement : KerML::Step;
}

behavior Save specializes UserAction {
    @Command; // This is valid.
    redefine step doAction {
        @Command; // This is valid.
    }
}
struct Options {
```

```
    @Command; // This is INVALID.
}
```

If the `metaclass` of a MetadataFeature is a direct or indirect specialization of *Metaobjects::SemanticMetadata* (see 8.16.2.3), then the `annotatedElements` must all be Types and the Feature *`SemanticMetadata::baseType`* must be bound to a value of type *KerML::Type* (see 8.17). Then each annotated Type shall implicitly specialize a Type determined from the *`baseType`* value as follows:

- If the annotated Type is neither a Classifier nor a Feature, then the annotated Type shall implicitly specialize the `baseType`.
- If the annotated Type is a Classifier and the `baseType` is a Classifier, then annotated Classifier shall implicitly subclassify the `baseType`.
- If the annotated Type is a Classifier and the `baseType` is a Feature, then the annotated Classifier shall implicitly subclassify each type of the `baseType`.
- If the annotated type is a Feature and the `baseType` is a Feature, then the annotated Feature shall implicitly subset the `baseType`.
- In all other cases, no implicit specialization is added.

When evaluated in a model-level evaluable expression, the cast operator **`as`** (see 7.4.8.2.1) may be used to cast a Feature referenced as its first operand to the actual reflective Metaclass value for this Feature, which may then be bound to the *`baseType`* Feature of *SemanticMetadata*.

```
behavior UserAction;
step userActions : UserAction[*] nonunique;

metaclass Command specializes SemanticMetadata {
    // The cast operation "userAction as KerML::Feature" has
    // type Feature, which conforms to the type Type of baseType.
    // Since userActions is a Step, the expression evaluates at
    // model level to a value of type KerML::Step.
    baseType = userActions as KerML::Feature;
}

// Save implicitly subclassifies UserAction (which is the
// type of userActions).
behavior Save {
    @Command;
}

// previousAction implicitly subsets userActions.
step previousAction[1] {
    @Command;
}
```

Release Note. This use of the cast operator for "meta-casting" is a workaround until a more general notation for reflection is introduced.

7.4.12.3 Abstract Syntax

7.4.12.3.1 Overview

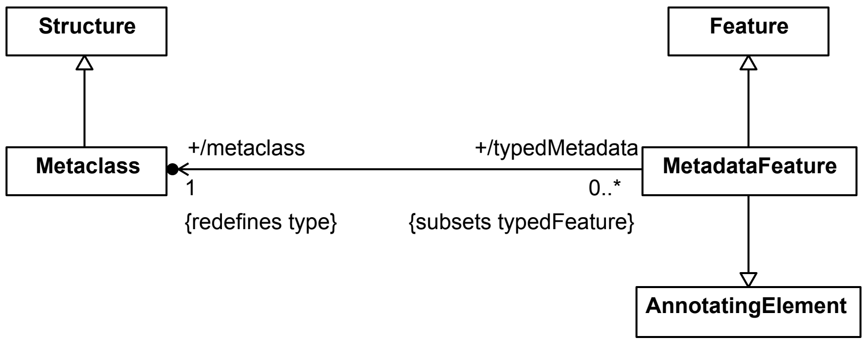

Figure 34. Metadata Annotation

7.4.12.3.2 Metaclass

Description

A Metaclass is a Structure used to type MetadataFeatures. It must subclassify, directly or indirectly, the base type *`Metadata`* from the Kernel Library.

General Classes

Structure

Attributes

None.

Operations

No operations.

Constraints

None.

7.4.12.3.3 MetadataFeature

Description

A MetadataFeature is a Feature that is an AnnotatingElement used to annotate another Element with metadata. It is typed by a Metaclass. All its `ownedFeatures` must redefine `features` of its `metaclass` and any feature bindings must be model-level evaluable.

A MetadataFeature must subset, directly or indirectly, the base MetadataFeature *`metadata`* from the Kernel Library.

General Classes

AnnotatingElement
Feature

Attributes

/metaclass : Metaclass {redefines type}

The `type` of this AnnotatingFeature, which must be a DataType.

Operations

No operations.

Constraints

None.

7.4.12.4 Semantics

Required Specializations of Model Library

1. Metaclasses shall directly or indirectly specialize *Metaobjects::Metaobject* (see 8.16.2.1).
2. MetadataFeatures shall directly or indirectly specialize *`Metaobjects::metaobjects`* (see 8.16.2.2), which means they shall be typed by Metaclasses.
3. If a MetadataFeature directly or indirectly specializes *Metaobjects::SemanticMetadata* (see 8.16.2.3), then
   - If one of the following holds:
     - the annotated Type is neither a Classifier nor a Feature
     - the annotated Type is a Classifier and the Type bound to the *`baseType`* Feature of the MetadataFeature is a Classifier
     - the annotated Type is a Feature and the Type bound to the *`baseType`* Feature of the MetadataFeature is a Feature

     then the annotated Type shall directly or indirectly specialize the Type referenced by the *`baseType`*.
   - If the annotated Type is a Classifier and the Type bound to the *`baseType`* Feature of the MetadataFeature is a Feature, then the annotated classifier shall directly or indirectly specialize each `type` of the Feature referenced by the *`baseType`*.

Metadata

As noted in 7.4.12.1, while MetadataFeatures are Features, they are defined only within a model and do not have instance-level semantics (i.e., they do not affect instances, as specified in Core – see 7.3.4). However, at a meta-level, a MetadataFeature can be treated as if the (reflective) Metaclasses of its `annotatedElements` where its `featuringTypes`. In this case, the MetadataFeature defines a map from its `annotatedElements`, as instances of their Metaclasses, to a single instance of its `metaclass`.

Further, a model-level evaluable Expression is simply an Expression that can be evaluated using metadata available within a model itself (see 7.4.7.2.2). If a model-level evaluable Expression is evaluated on such metadata according to the regular semantics of Expressions, then the result will be the same as the static evaluation of the Expression within the model. Therefore, if a MetadataFeature is instantiated as above, the binding of its features to the results of evaluating the model-level evaluable Expressions given as `featureValues` can be interpreted according to the regular semantics of FeatureValues (see 7.4.10) and BindingConnectors (see 7.4.5).

When a `value` Expression is model-level evaluated (as described in 7.4.8.4), its target is the MetadataFeature that owns the associated `metadataFeature`. This means that the `value` Expression for a nested Feature of a `MetadataFeature` may reference other Features of the `MetadataFeature`, as well as Features with no `featuringTypes` or *`Anything`* as a `featuringType`.

7.4.13 Packages

7.4.13.1 Packages Overview

Packages are Namespaces used to group Elements, without any instance-level semantics (as opposed to Types, which are Namespaces with classification semantics, see 7.3.2). They might also have one or more `filterConditions` for selecting a subset of its `importedMemberships`. A `filterCondition` is a Boolean-valued, model-level evaluable Expression (see 7.4.8) related to a Package by an ElementFilterMembership. Each `filterCondition` of a Package shall result in *`true`* when model-level evaluated (see 7.4.8.4) for any imported `member` of the Package as described in 7.4.13.4.

A `filterCondition` can operate on metadata on Elements (see 7.4.12), such as checking whether has a MetadataFeature of a particular Type and accessing the values of the `features` of a MetadataFeature. For the purposes of `filterCondition` Expressions, every Element is also considered to have an implicit MetadataFeature that is typed by a Metaclass from the reflective library model of the KerML abstract syntax. This enables `filterConditions` to test for the abstract syntax metaclass of an Element and to access the values of abstract syntax meta-attributes.

Implementation Note. The implemented *KerML* library model currently contains the declaration of all abstract syntax Metaclasses, but does not yet include any meta-attributes.

7.4.13.2 Concrete Syntax

```
Package : Package =
    PackageDeclaration(this) PackageBody(this)

PackageDeclaration (p : Package) =
    'package' Identification(p)

PackageBody (p : Package) =
      ';'
    | '{' ( NamespaceBodyElement(p)
          | p.ownedRelationship += ElementFilterMember )*
      '}'

ElementFilterMember : ElementFilterMembership =
    MemberPrefix(this)
    'filter' condition = OwnedExpression ';'
```

A Package is notated like a generic Namespace, but using the keyword **`package`** instead of **`namespace`**.

```
package AddressBooks {
    datatype Entry {
        feature name: String;
        feature address: String;
    }
    struct AddressBook {
        composite feature entries[*]: Entry;
    }
}
```

In addition, a Package body may contain one or more members that give `filterConditions` for the Package. These are notated using the keyword **`filter`** followed by a Boolean-valued, model-level evaluable Expression.

```
package Annotations {
    datatype ApprovalAnnotation {
```

```
        feature approved : Boolean;
        feature approver : String;
        feature level : Natural;
    }
    ...
}

package DesignModel {
    import Annotations::*;
    struct System {
         @ApprovalAnnotation {
            approved = true;
            approver = "John Smith";
            level = 2;
        }
    }
    ...
}

package UpperLevelApprovals {
    // This package imports all direct or indirect members
    // of the DesignModel package that have been approved
    // at a level greater than 1.
    import DesignModel::**;
    filter @Annotations::ApprovalAnnotation &&
           Annotations::ApprovalAnnotation::approved &&
           Annotations::ApprovalAnnotation::level > 1;
}
```

Note that a `filterCondition` in a Package will filter *all* imports of that Package. That is why full qualification is used for *`Annotations::ApprovalAnnotation`* above, since an imported elements of the *Annotations* Package would be filtered out by the very `filterCondition` in which the elements are intended to be used. This may be avoided by combining one or more `filterConditions` with a specific import, using the filtered Import notation defined in 7.2.4.2.2.

```
package UpperLevelApprovals {
    // Recursively import all annotation data types and all
    // features of those types.
    import Annotations::**;

    // The filter condition for this import applies only to
    // elements imported from the DesignModel package.
    import DesignModel::**[@ApprovalAnnotation && approved && level > 1];
}
```

The *KerML* library package contains a complete model of the KerML abstract syntax represented in KerML itself. When a filterCondition is evaluated on an Element, abstract syntax metadata for the Element can be tested as if the Element had an implicit AnnotatingFeature typed by the Type from the *KerML* package corresponding to the metaclass of the Element.

```
package PackageApprovals {
    import Annotations::*;
    import KerML::*;

    // This imports all structures from the DesignModel that have
    // at least one owned feature and have been marked as approved.
    import DesignModel::**[@Structure &&
                           @Structure::ownedFeature != null &&
```

```
                                        @ApprovalAnnotation &&
                                        ApprovalAnnotation::approved];
    }
```

7.4.13.3 Abstract Syntax

7.4.13.3.1 Overview

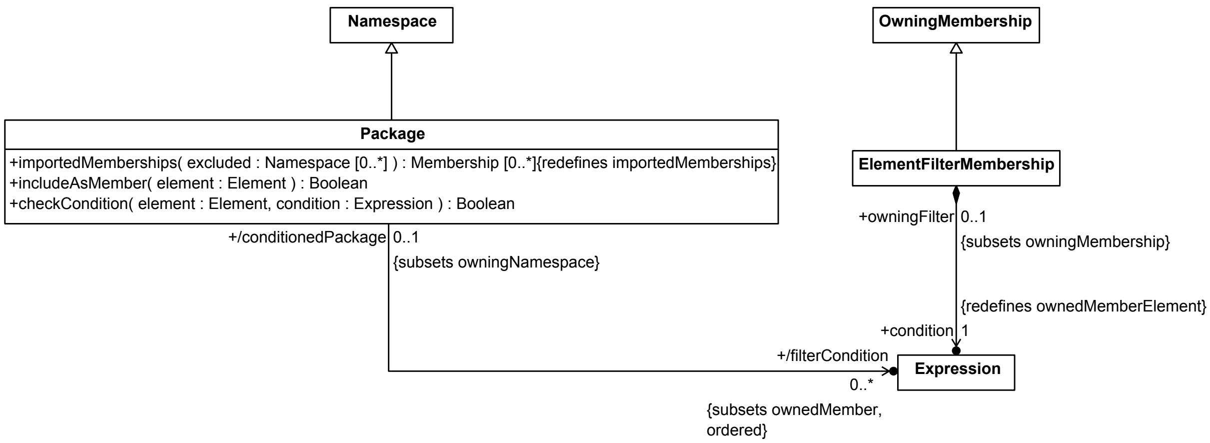

Figure 35. Packages

7.4.13.3.2 ElementFilterMembership

Description

ElementFilterMembership is a Mambership between a Namespace and a model-level evaluable Boolean Expression, asserting that imported `members` of the Namespace should be filtered using the `condition` Expression. A general Namespace does not define any specific filtering behavior, but such behavior may be defined for various specialized kinds of Namespaces.

General Classes

OwningMembership

Attributes

condition : Expression {redefines ownedMemberElement}

The model-level evaluable Boolean Expression used to filter the `members` of the `membershipOwningNamespace` of this ElementFilterMembership.

Operations

No operations.

Constraints

elementFilterIsBoolean

The `result` Feature of the `condition` Expression must have *ScalarValues::Boolean* as a `type`.

elementFilterIsModelLevelEvaluable

The `condition` Expression must be model-level evaluable.

```
condition.isModelLevelEvaluable
```

7.4.13.3.3 Package

Description

A Package is a Namespace used to group Elements, without any instance-level semantics. It may have one or more model-level evaluable `filterCondition` Expressions used to filter its `importedMemberships`. Any imported `member` must meet all of the `filterConditions`.

General Classes

Namespace

Attributes

/filterCondition : Expression [0..*] {subsets ownedMember, ordered}

The model-level evaluable Boolean Expressions used to filter the `members` of this Package, derived as those `ownedMembers` of the Package that are owned via ElementFilterMembership.

Operations

checkCondition(element : Element,condition : Expression) : Boolean

Model-level evaluate the given `condition` Expression with the given `element` as its target. If the result is a LiteralBoolean, return its `value`. Otherwise return `false`.

```
body: let results: Sequence(Element) = condition.evaluate(element) in
    result->size() = 1 and
    results->at(1).oclIsKindOf(LiteralBoolean) and
    results->at(1).oclAsType(LiteralBoolean).value
```

importedMemberships(excluded : Namespace [0..*]) : Membership [0..*]

Exclude Elements that do not meet all the `filterConditions`.

```
body: self.oclAsType(Namespace).importedMemberships(excluded)->
    select(m | self.includeAsMember(m.memberElement))
```

includeAsMember(element : Element) : Boolean

Determine whether the given `element` meets all the `filterConditions`.

```
body: let metadataAnnotations: Sequence(AnnotatingElement) =
    element.ownedAnnotation.annotatingElement->
        select(oclIsKindOf(AnnotatingFeature)) in
    self.filterCondition->forAll(cond |
        metadataAnnotations->exists(elem |
            self.checkCondition(elem, cond)))
```

Constraints

packageImportVisibility

The `ownedImports` of a Package must not have a `visibility` of `protected`.

```
ownedImport->forAll(visibility <> VisibilityKind::protected)
```

packageOwnedMembershipVisibility

The `ownedMemberships` of a Package must not have a `visibility` of `protected`.

```
ownedMembership->forAll(visibility <> VisibilityKind::protected)
```

7.4.13.4 Semantics

Packages do not have semantics (they do not affect instances).

The `filterConditions` of a Package are model-level evaluable Expressions that are evaluated as described in 7.4.8.4. All `filterConditions` are checked against every Membership that would otherwise be imported into the Package if it had no `filterCondition`. A Membership shall be imported into the Package if and only if every `filterCondition` evaluates to true either with no target Element, or with any AnnotatingFeature of the `memberElement` of the Membership as the target Element.

Implementation Note. As of 2020-01, an AnnotatingFeature must be owned by the imported Element in order to be accessed when evaluating a `filterCondition`.

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 *Performance* (see 7.3.1.1 and the Semantics subclauses in Clause 7). The library can be specialized for particular applications, such as systems.

The Kernel Model Library is organized into a single KerML Package called *KernelLibrary.* Each library model is contained in a subpackage of *KernelLibrary*, as described in a subsequent subclause of this Clause. The following are the major areas covered in the Kernel Model Library.

1. The *Base* library model (see 8.2) begins the Specialization hierarchy for all KerML Types, including the most general Classifier *Anything* and the most general Feature `things`. It also contains the most general DataType *DataValue* and its corresponding Feature `dataValues`. The *Links* library model (see 8.3) specializes *Base* to provide the semantics for Associations between things.
2. The *Occurrences* library model (see 8.4) introduces *Occurrence*, the most general Class of things that exist or happen in time and space, as well as the basic temporal Associations between them. The *Objects* library model (see 8.5) specializes *Occurrences* to provide a model of *Objects* and *LinkObjects*, giving semantics to Structures and AssociationStructures, respectively. The *Performances* library model (see 8.6) specializes *Occurrences* to provide a model of *Performances* and *Evaluations*, giving semantics to Behaviors and Expressions, respectively. Temporal associations can be used by Successions 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 8.7) models asynchronous flow of items between *Occurrences,* giving semantics to Interactions and ItemFlows. The *FeatureAccessPerformances* (see 8.8) defines specialized *Performances* for access and modifying the values of features at specific points in time.
3. The *ControlPerformances, TransitionPerformances* and *StatePerformances* library models (see 8.9, 8.10, and 8.11) provide for coordination of multiple *Performances* to carry out some task by using them as `types` of Steps in an overall containing Behavior. KerML does not provide syntax specific to these library elements (e.g., KerML does not have any "control node" or "state machine" syntax), though it is expected that other languages built on KerML and these library models can add syntax as needed by their applications.
4. The *ScalarValues* and *Collections* model libraries (see 8.18 and 8.19) provide commonly needed primitive and collection DataTypes. Additional library models (see 8.21 through 8.36) provide Functions that operate on library DataTypes (and others specialized from them, see below). The KerML operator and sequence expression notations translate to invocations of some of these library Functions. It is expected that other languages built on KerML will provide additional domain models as needed by their applications, which can include specializations of the library Functions for domain-specific DataTypes. The same KerML concrete syntax for Expressions notation can be used with these specialized Functions and Datatypes, extended with domain-specific semantics.

The normative representation of all library models is in the textual concrete syntax, as provided in machine-readable files associated with this specification document.

Implementation Note. As of the 2021-10 release, the pilot implementation does not include the containing *KernelLibrary* package. Instead, all Kernel Model Library packages are visible in the global namespace.

8.2 Base

8.2.1 Base Overview

This library model begins the Specialization hierarchy for all KerML Types (see 7.3.2.1), starting with the most general Classifier *Anything*, the `type` of the most general Feature *`things`*, which classify everything in the modeled universe and the relations between them, respectively. Being the most general library elements for their metaclasses means all Classifiers and Features in models, including in libraries, specialize them, respectively. They are specialized into most general DataType *DataValue*, the `type` of *`dataValues`*, the most general Feature typed by DataTypes, respectively (see 7.4.2.1). *DataValues* are *Anything* that can only be distinguished by how they are related to other things (via Features and Assocations). These are are further specialized into *Natural* and *`naturals`*, respectively, an extension for mathematical natural numbers (integers zero and greater) extended with a number greater than all the integers ("infinity"), but treated like one, notated as `*` (see 8.18.1). The Feature *`self`* of *Anything* relates each thing in the universe to itself only (see *SelfLinks* in 8.3.1).

8.2.2 Elements

8.2.2.1 Anything

Element

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 `type` for `things`, the most general Feature. Since FeatureTyping is a kind of Generalization, this means that Anything is also a generalization of `things`.

General Types

None.

Features

myself : Anything {subsets myselfSameLife}

The target end of a SelfLink.

myselfSameLife : Anything [1..*] {redefines toSources}

The target end of a SelfLifeLink.

self : Anything {subsets selfSameLife}

The source end of a SelfLink.

selfSameLife : Anything [1..*] {redefines toTargets}

The source end of a SelfLifeLink.

Constraints

None.

8.2.2.2 DataValue

Element

DataType

Description

A DataValue is Anything that can only be distinguished by how it isrelated to other things (via Features). DataValue 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 Types

Anything

Features

None.

Constraints

None.

8.2.2.3 dataValues

Element

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 Types

DataValue
things

Features

None.

Constraints

None.

8.2.2.4 naturals

Element

Feature

Description

General Types

Natural
dataValues

Features

None.

Constraints

None.

8.2.2.5 SelfSameLifeLink

Element

Association

Description

SelfLifeLinks are all and only BinaryLinks where the sourceParticipant and targetParticipant are either

- Occurrences (which might be lives) that are portions of the same life, or
- Data values that are equal.

General Types

BinaryLink

Features

sourceDataValue : DataValue [0..1] {subsets source}

Same as the sourceParticipant when it is a data value.

sourceOccurrence : Occurrence [0..1] {subsets source}

Same as the sourceParticipant when it is an occurrence.

targetDataValue : DataValue [0..1] {subsets target}

Same as the targetParticipant when it is a data value.

targetOccurrence : Occurrence [0..1] {subsets target}

Same as the targetParticipant when it is an occurrence.

Constraints

None.

8.2.2.6 things

Element

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 Types

Anything

Features

None.

Constraints

None.

8.3 Links

8.3.1 Links Overview

This library model introduces the most general Association *Link*, the type of *`links`*, the most general Feature typed by Associations (see 7.4.4.1). The *`participant`* Feature of *Link* is the most general `associationEnd`, identifying the things being linked by (at the "ends" of) each *Link* (exactly one thing per end, which might be the same things). *Link* is specialized into *BinaryLink*, the most general Association with exactly two `associationEnds`, *source* and *target*, which subset *`participant`* and identify the two things linked, which might be the same thing. *BinaryLink* is the `type` of *`binaryLinks`*, the most general Feature typed by binary Associations. They are specialized into *SelfLink* and *`selfLinks`*, respectively, for links that have the same thing on both ends, identified by *`thisThing`* and *`thatThing`*, redefining *`source`* and *`target`*, respectively. These are used by BindingConnectors to specify that Features have the same values (see 7.4.5.1). *SelfLinks* are not in time or space (they are not Occurrences, see 8.5.1).

8.3.2 Elements

8.3.2.1 BinaryLink

Element

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 Types

Link

Features

participant : Anything {redefines participant, ordered, nonunique}

The participants of this BinaryLink, which are restricted to be exactly two.

source : Anything {subsets participant}

The participant that is the source of this BinaryLink.

target : Anything {subsets participant}

The participant that is the target of this BinaryLink.

toSources : Anything [0..*]

The end Feature of this BinaryLink corresponding to the `sourceParticipant`.

toTargets : Anything [0..*]

The end Feature of this BinaryLink corresponding to the `targetParticipant`.

Constraints

None.

8.3.2.2 binaryLinks

Element

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 Types

links
BinaryLink

Features

[no name] : Anything

[no name] : Anything

Constraints

None.

8.3.2.3 Link

Element

Association

Description

Link 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 Types

Anything

Features

participant : Anything [2..*] {ordered, nonunique}

The participants that are associated by this Link.

Constraints

None.

8.3.2.4 links

Element

Feature

Description

`links` is a specialization of `things` 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 Types

Link
things

Features

None.

Constraints

None.

8.3.2.5 SelfLink

Element

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 or indirectly).

General Types

BinaryLink
SelfSameLifeLink

Features

sourceParticipant : Anything {subsets targetParticipant, redefines source}

The source participant of this SelfLink, which must be the same as the target participant.

targetParticipant : Anything {subsets sourceParticipant, redefines target}

The target participant of this SelfLink, which must be the same as the source participant.

Constraints

None.

8.3.2.6 selfLinks

Element

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 Types

SelfLink
binaryLinks

Features

[no name] : Anything

[no name] : Anything

Constraints

None.

8.4 Occurrences

8.4.1 Occurrences Overview

Occurrences

This library adds the most general time and space model, starting with the most general Class *Occurrence*, which classifies *Anything* that takes up time and space, and *`occurrences`*, the most general Feature typed by Classes (see 7.4.2.1). *Occurrences* divide into *Objects* and *Performances* (see 8.5.1 and 8.6.1, respectively), corresponding to Classes dividing into Structures and Behaviors (see Structures and Behaviors, respectively). This subclause covers what is in common between *Objects* and *Performances*.

Temporal Associations

Occurrences are related in time by *HappensLinks*, in particular *HappensDuring* and *HappensBefore*, which indicate when one occurrence happens or exists within the time taken by another, or they happen or exist separately in time, respectively. The `suboccurrences` of *Occurrences* are ones that *HappenDuring* them, while the `predecessors` and `successors` of *Occurrences* are those that *HappenBefore* them and after them (those that they *HappenBefore*), respectively. *HappensJustBefore Links* are *HappensBefore Links* between *Occurrences* where there is no possibility of other *Occurrences* happening in the time between them. The `immediatePredecessors` and `immediateSuccessors` of *Occurrences* are those that *HappenJustBefore* them and just after them (those that they *HappenJustBefore*), respectively. *HappensLinks* to do not take up time or space, they are temporal relations between things that do (they are disjoint with *LinkObject*, see 8.5.1).

Occurrences cannot be linked by both *HappensDuring* and *HappensBefore*. They also cannot *HappenBefore* themselves, but always *HappenDuring* themselves. *Occurrences* that *HappenDuring* each other both ways (circularly) happen or exist at the same time, which is provided for convenience by *HappensWhile*, a specialization of *HappenDuring*. *HappensDuring* and *HappensBefore* also constrain each other. When an *Occurrence HappensBefore* another, all *Occurrences* that *HappenDuring* the earlier one (including itself) also *HappenBefore* those that *HappenDuring* the later one (including itself).

Portions and Time Slices

It is useful to consider *Occurrences* during only some of the time they happen or exist, but including all the space they take up during that time. These are also *Occurrences*, because they take up time and space, and are `timeSlicesOf` the "larger" *Occurrences* that they are `portionsOf` (`portions` do not necessarily take up all time or space of their *Occurrence*). *Occurrences* that are `timeSlicesOf` others are the same "thing" as their larger *Occurrences*, just considered for a smaller period of time (likewise for `portions`), during which they might have Feature values and Links to other things peculiar to that smaller period. They must be classified the same way as the *Occurrences* they are `timeSlicesOf` (and `portionsOf`), or more specialized.

Occurrences are always `timeSlicesOf` (and `portionsOf`) themselves. *Occurrences* that are only `timeSlicesOf` (and `portionsOf`) of themselves are *Lives* (classifed by the library Class *Life*). *Lives* take up the entire time and space of a thing that happens or exists. *Occurrences* have the same *Life* as those they are *portionsOf*, identified by *portionOfLife*.

The `snapShots` of *Occurrences* are `timeSlices` that takes zero time. The earliest `snapShot` of an *Occurrence* is its `startShot`, the latest is its `endShot`. All the others happen during its `middleTimeSlice`. *Occurrences* with `startShot` the same as their `endShot` take no time, have no `middleTimeSlice`, and vice-versa.

SelfSameLifeLinks include *SelfLinks* (*Links* between *Anything* and themselves, see 8.3.1), as well as *Links* between *Occurrences* that are *portionsOf* the same Life (have the same *portionOfLife*).

Spatial Associations

Release Note. To be documented.

8.4.2 Elements

8.4.2.1 HappensBefore

Element

Association

Description

HappensBefore is a *Withoutassociation linking an* `earlierOccurrence` *to a* `laterOccurrence`*, indicating that the Occurrences do not overlap in time (not necessarily in space, see OutsideOf; none of their* `snapshots` *happen at the same time), and the* `earlierOccurrence` *happens first. This means noOccurrence HappensBefore itself. Every Occurrence that HappensDuring the* `earlierOccurrence` *(including itself) also HappensBefore every Occurrence that HappensDuring the* `laterOccurrence` *(including itself).*

General Types

HappensLink
Without

Features

earlierOccurrence : Occurrence {redefines separateOccurrenceToo}

The participant that happens earlier than (before) the other participant.

laterOccurrence : Occurrence {redefines separateOccurrence}

The participant that happens later than (after) the other participant.

Constraints

None.

8.4.2.2 happensBeforeLinks

Element

Feature

Description

`happensBeforeLinks` 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 Types

HappensBefore
binaryLinks

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

8.4.2.3 HappensDuring

Element

Association

Description

HappensDuring links its `shorterOccurrence` to its `longerOccurrence`, indicating that the `shorterOccurrence` completely overlaps the `longerOccurrence` in time (not necessarily in space, see *InsideOf*; all `snapshots` of the `shorterOccurrence` happen at the same time as some `snapshot` of the `longerOccurrence`). This means every *Occurrence/ HappensDuring itself and that HappensDuring is transitive. Every Occurrence that HappensBefore the* `longerOccurrence` *also HappensBefore the* `shorterOccurrence`*. The* `shorterOccurrence` *also HappensBefore every Occurrence that the* `longerOccurrence` *does.*

General Types

HappensLink

Features

longerOccurrence : Occurrence {redefines targetOccurrence}

The participant in this HappensDuring Link that takes up more (or equal) time than the other.

shorterOccurrence : Occurrence {redefines sourceOccurrence}

The participant in this HappensDuring Link that takes up less (or equal) time than the other.

Constraints

None.

8.4.2.4 HappensJustBefore

Element

Association

Description

HappensJustBefore is *HappensBefore* asserting that there is no possibility of other *Occurrences* happening in the time between the *`earlierOccurrence`* and *laterOccurrence*.

General Types

HappensBefore

Features

None.

Constraints

None.

8.4.2.5 HappensLink

Element

Association

Description

General Types

BinaryLink

Features

happensSource : Occurrence [0..*] {subsets toSources}

happensTarget : Occurrence [0..*] {subsets toTargets}

sourceOccurrence : Occurrence {redefines source}

targetOccurrence : Occurrence {redefines target}

Constraints

None.

8.4.2.6 HappensWhile

Element

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 and that *HappensWhile* is transitive.

General Types

HappensDuring

Features

thatOccurrence : Occurrence {redefines longerOccurrence}

thisOccurrence : Occurrence {redefines shorterOccurrence}

Constraints

None.

8.4.2.7 InsideOf

Element

Association

Description

InsideOf is a BinaryLink between its `smallerSpace` and `largerSpace`, indicating that the `largerSpace` completely overlaps the `smallerSpace` in space (not necessarily in time, see HappensDuring; all four dimensional points of the `smallerSpace` are in the spatial extent of the `largerSpace`). This means every *Occurrence/ is InsideOf itself and that InsideOf is transitive.*

General Types

BinaryLink

Features

largerSpace : Occurrence {redefines target}

The participant in this InsideOf Link that takes up more (or equal) space than the other.

smallerSpace : Occurrence {redefines source}

The participant in this InsideOf Link that takes up less (or equal) space than the other.

Constraints

None.

8.4.2.8 Life

Element

Class

Description

Life is the class of Occurrences that are "maximal portions". That is, they are only portions of themselves.

General Types

Occurrence

Features

portion : Occurrence [1..*]

Occurrences that are portions of this Life, including at least this Life.

Constraints

None.

8.4.2.9 Occurrence

Element

Class

Description

An Occurrence is Anything that happens over time and space (the four physical dimensions). Occurrences can be portions of another Occurrence within time and space, including slices in time, leading to snapshots that take zero time.

General Types

Anything

Features

difference : Occurrence [0..1]

A (nested) feature of `differencesOf` identifying an Occurrence that is the `intersectionsOf` of the Occurrences identified by `interdiff` (`minuend` and `interdiff.notSubtrahend`).

differencesOf : OrderedSet [0..*]

Ordered sets of Occurrences, where the time and space taken by first Occurrence in each set (`minuend`) that is not in the time and space taken by the remaining Occurrences (`subtrahend`, resulting in `difference`) is the same as taken by this Occurrence (all four dimensional points in the `minuend` that are not in any `subtrahend` are at the same time and space as those in this Occurrence).

elements : Occurrence [0..*]

A nested feature of `unionsOf`, `intersectionsOf`, and `differencesOf` for the elements of each of their (Ordered)Sets separately.

endShot : Occurrence {subsets snapshot}

The `snapshot` of this Occurrence that `happensAfter` all its other `snapshots`.

endShotOf : Occurrence [0..*] {subsets snapshotOf}

Occurrences of which this Occurrence is the `endShot`.

happensDuring : Occurrence [1..*] {subsets happensTarget}

Occurrences that completely overlap this one in time (not necessarily in space, see `insideOf`; they start when this one does or earlier and end when this one does or later), including this one.

happensWhile : Occurrence [1..*] {subsets happensDuring}

Occurrences that start and end at the same time as this one.

happensWhile[1] : Occurrence [1..*] {subsets timeEnclosedOccurrences}

Occurrences that `happenWhile` this one does (Occurrences that start and end at the same time as this one).

immediatePredecessors : Occurrence [0..*] {subsets predecessors}

Occurrences that *HappensJustBefore* this one (*Occurrences* that *HappensBefore* this one, with no possibility of other *Occurrences* happening in the time between them).

immediateSuccessors : Occurrence [0..*] {subsets successors}

Occurrences that this one *HappensJustBefore* (*Occurrences* that this one *HappensBefore*, with no possibility of other *Occurrences* happening in the time between them).

incomingTransfer : Transfer [0..*]

incomingTransferToSelf : Transfer [0..*] {subsets incomingTransfer}

Transfers for which this Occurrence is the `targetParticipant`.

innerSpaceDimension : Natural

The number of variables need to identify space points in this Occurrence, from 0 to 3, without regard to higher dimensional spaces it might be emedded in. For example, the `innerSpaceDimension` of a curve is 1, even if it twists in three dimensions, see `outerSpaceDimension`.

insideOf : Occurrence [1..*] {subsets toTargets}

Occurrences that completely overlap this one in space (not necessarily in time, see `happensDuring`), including this one.

interdiff : Set [0..*]

A (nested) feature of `differencesOf` identifying a set that includes its `minuend` and all Occurrences that are not in its `subtrahend`.

intersection : Occurrence [0..1]

A (nested) feature of `intersectionsOf` identifying an Occurrence that a) is completely `within` (the space and time of) all `intersectionsOf` `elements`, and b) satisfies the conditions of the same `element`'s `nonIntersection`.

intersectionsOf : Set [0..*]

Sets of Occurrences, where the time and space taken in common between the Occurrences in each set (`intersectionsOf::intersection`) is at the same as taken by this Occurrence (all four dimensional points common to the Occurrences in each set are at the same time and space as those in this Occurrence).

/isClosed : Boolean

True if this Occurrence has a `spaceBoundary`, false otherwise.

middleTimeSlice : Occurrence [0..1] {subsets timeSlices}

`timeSlice` of this Occurrence that takes all of the time between its `startShot` and code>endShot. Occurrences do not have `middleTimeSlice` if their `startShot` is the same as their `endShot` (such as being a `snapShot` of another Occurrence), otherwise they do.

middleTimeSliceOf : Occurrence [0..1] {subsets timeSliceOf}

Occurrences of which this Occurrence is a `middleTimeSlice`.

minuend : Occurrence [0..1] {subsets }

A (nested) feature of `differencesOf` that identifies the first Occurrence in its `elements`.

myself : Occurrence {subsets timeSliceOf, spaceSliceOf, redefines myself}

nonIntersection : Occurrence [0..*] {subsets spaceTimeEnclosedPoints}

A nested feature of `intersectionsOf.elements` identifying all the `spaceTimeEnclosedPoints` of each `element` that are not identified by `intersection`. These must be `without` (separate in space or time from) at least one other `element`.

notSubtrahend : Occurrence [0..*]

A (nested) feature of `differencesOf.interdiff` identifying all Occurrences that are not identified by the `subtrahend` in each value `differencesOf` separately.

outerSpaceDimension : Natural [0..1]

For Occurrences of `innerSpaceDimension` 1 or 2, the number of variables needed to identify their space points in higher dimensional spaces they might be embedded in, from the `innerSpaceDimension` to 3. For example , an `outerSpaceDimension` 3 for a curve indicates it twists in three dimensions. An `outerSpaceDimension` equal to `innerSpaceDimension` indicates the occurrence is spatially straight (`innerSpaceDimension` 1 embedded in 2 or 3 dimensions) or flat (`innerSpaceDimension` 2 embedded in 3 dimensions).

outgoingTransfer : Transfer [0..*]

outgoingTransferFromSelf : Transfer [0..*] {subsets outgoingTransfer}

Transfers for which this Occurrence is the `sourceParticipant`.

outsideOf : Occurrence [0..*] {subsets without}

Occurrences that are completely separate from this one in space (not necessarily in time, see `successors`).

outsideOfToo : Occurrence [0..*] {subsets withoutToo}

Occurrences that are completely separate from this one in space (not necessarily in time, see `predecessors`).

portion : Occurrence [1..*] {subsets spaceTimeEnclosedOccurrences}

All occurrences `within` this one that are considered the same thing occurring (same `portionOfLife`), including this one.

portionOf : Occurrence [1..*] {subsets within}

All occurrences that this one is `within`that are considered the same thing occurring (same `portionOfLife`), including this one.

portionOfLife : Life

The Life of which this Occurrence is a `portion`.

predecessors : Occurrence [0..*] {subsets withoutToo}

Occurrences that are completely separate from this one in time (not necessarily in space, see `outsideOfToo`) and that happen before this one (end earlier than this one starts).

self : Occurrence {subsets timeSlices, spaceSlices, redefines self}

This Occurrence (related to itself via a SelfLink).

snapshot : Occurrence [1..*] {subsets timeSlices}

All `timeSlices` of this Occurrence that happen at a single instant of time (zero duration).

snapshotOf : Occurrence [0..*] {subsets timeSliceOf}

Occurrences of which this Occurrence is a `snapshot`.

spaceBoundary : Occurrence [0..1] {subsets spaceShots}

A `spaceShot` of this Occurrence that is not among those of its `spaceInterior`, which it must be `outsideOf`. Must have no `spaceBoundary` (`isClosed` = true).

spaceBoundaryOf : Occurrence [0..*] {subsets spaceShotOf}

An Occurrence this one is the `spaceBoundary` of.

spaceEnclosedOccurrences : Occurrence [1..*] {subsets toSources}

Occurrences that this one completely overlaps in space (not necessarily in time, see `timeEnclosedOccurrences`), including this one.

spaceInterior : Occurrence [0..1] {subsets spaceSlices}

A `spaceSlice` of this Occurrence that includes all its `spaceShots` except the `spaceBoundary`, which must exist and be `outsideOf` it. The `spaceInterior` must be of the same `innerSpaceDimension` as this Occurrence, except if it is zero, whereupon there is no `spaceInterior`.

spaceInteriorOf : Occurrence [0..1] {subsets spaceSliceOf}

An Occurrence this one is the `spaceInterior` of.

spaceShotOf : Occurrence [1..*] {subsets spaceSliceOf}

All `spaceSlicesOf` this Occurrence that are of a higher `innerSpaceDimension` than this Occurrence.

spaceShots : Occurrence [1..*] {subsets spaceSlices}

All `spaceSlices` of this Occurrence that are of a lower `innerSpaceDimension` than it.

spaceSliceOf : Occurrence [1..*] {subsets portionOf}

An Occurrence this one is a `spaceSlices` of.

spaceSlices : Occurrence [1..*] {subsets portion}

All `portions` of this Occurrence that extend for exactly the same time and some or all the space, relative to spatial location of this Occurrence. This means every Occurrence is a *spaceSlice* of itself.

spaceTimeCoincidentOccurrences : Occurrence [1..*] {subsets spaceTimeEnclosedOccurrences}

spaceTimeEnclosedOccurrences : Occurrence [1..*] {subsets spaceEnclosedOccurrences, timeEnclosedOccurrences}

All `timeEnclosedOccurrences` of this one that are `insideOf` it, including itself.

spaceTimeEnclosedPoints : Occurrence [1..*] {subsets spaceTimeEnclosedOccurrences}

All `spaceTimeEnclosedOccurrences` of this one that take up no time or space (`innerSpaceDimension` 0 and `startShot` the same as `endShot`).

startShot : Occurrence {subsets snapshot}

The `snapshot` of this Occurrence that `happensBefore` all its other `snapshots`.

startShotOf : Occurrence [0..*] {subsets snapshotOf}

Occurrences of which this Occurrence is the `startShot`.

subtrahend : Occurrence [0..*] {subsets }

A (nested) feature of `differencesOf` that identifies all the Occurrences in its `elements` except the first one.

successors : Occurrence [0..*] {subsets without}

Occurrences that are completely separate from this one in time (not necessarily in space, see `outsideOf`) and that happen after this one (start later than this one ends).

timeEnclosedOccurrences : Occurrence [1..*] {subsets happensSource}

Occurrences that this one completely overlaps in time (not necessarily in space, see `inside`; they start at the same time or later and end at the same time or earlier), including this one.

timeSliceOf : Occurrence [1..*] {subsets portionOf}

Occurrences of which this one is a `timeSlice`, including this one.p

timeSlices : Occurrence [1..*] {subsets portion}

`portions`that extend for some or all the time of this Occurrence, but all its space during that time, including itself.

union : Occurrence [0..1]

A (nested) feature of `unionsOf` identifying an Occurrence with a) `spaceTimeEnclosedOccurrences` including all those identified by a `unionsOf element`, and b) all the Occurrence's `spaceTimeEnclosedPoints within` (the space and time of) at least one of the `elements`.

unionsOf : Set [0..*]

Sets of Occurrences, where the time and space taken by all the Occurrences in each set together (`unionsOf::union`) is the same as taken by this Occurrence (all four dimensional points in the Occurrences of each set are at the same time and space as those of this Occurrence).

within : Occurrence [1..*] {subsets happensDuring, insideOf}

All Occurrences that this one `happensDuring` and is `insideOf`, including this one.

withinBoth : Occurrence [1..*] {subsets within}

without : Occurrence [0..*] {subsets toTargets}

All Occurrences that are `successors` of this one and are `outsideOf` it.

withoutToo : Occurrence [0..*] {subsets toSources}

All Occurrences that are `predecessors`of this one and are `outsideOfToo` it.

Constraints

None.

8.4.2.10 occurrences

Element

Feature

Description

`occurrences` is a specialization of `things` restricted to type Occurrence. It is the most general feature typed by Occurrence. All other Features typed by Occurrence or its specializations (in libraries or user models) specialize it (directly or indirectly).

General Types

things
Occurrence

Features

None.

Constraints

None.

8.4.2.11 OutsideOf

Element

Association

Description

OutsideOf is a *Without*association linking its `separateSpaceToo` and its `separateOccurrence`, indicating that these *Occurrences* do not overlap in space (not necessarily in time, see *HappensBefore*; no four dimensional points of the *Occurrences* are in the spatial extent of both of them). This means no *Occurrence is OutsideOf itself.*

General Types

Without

Features

separateSpace : Occurrence {redefines separateOccurrence}

The second participant in this OutsideOf Link.

separateSpaceToo : Occurrence {redefines separateOccurrenceToo}

The first participant in this OutsideOf Link.

Constraints

None.

8.4.2.12 PortionOf

Element

Association

Description

*PortionOf*is a *Within* that links its `portionOccurrence` to its `portionedOccurrence`, indicating they are considered the same thing occurring (same `portionOfLife`), but with the `portionOccurrence` potentially taking up less time and space than the `portionedOccurrence`. This means every *Occurrence/ is a PortionOf itself. The* *`innerSpaceDimension` of `portionOccurrence` is the same or lower than of the `portionedOccurrence`.*

General Types

Within

Features

portionedOccurrence : Occurrence {redefines largerOccurrence}

The participant in this PortionOf Link that is the `largerOccurrence`.

portionOccurrence : Occurrence {redefines smallerOccurrence}

The participant in this PortionOf Link that is the `smallerOccurrence`.

Constraints

None.

8.4.2.13 SnapshotOf

Element

Association

Description

*SnapshotOf*is a *TimeSliceof* that links its `snapshotOccurrence` to its `snapshottedOccurrence`, indicating that `snapshotOccurrence` takes not time (startShot and endShot are the same).

General Types

TimeSliceOf

Features

snapshotOcccurrence : Occurrence {redefines timeSliceOccurrence}

The participant in this SnapshotOf Link that is the `timeSliceOccurrence`.

snapshottedOccurrence : Occurrence {redefines timeSlicedOccurrence}

The participant in this SnapshotOf Link that is the `timeSlicedOccurrence`.

Constraints

None.

8.4.2.14 SpaceShotOf

Element

Association

Description

SpaceShotOf is a *SpaceSliceOf* that links its `spaceShotOccurrence` to its `spaceSnapshottedOccurrence`, indicating the `spaceShotOccurrence` is of a lower `innerSpaceDimension` than the `spaceShottedOccurrence`.

General Types

SpaceSliceOf

Features

spaceShotOccurrence : Occurrence {redefines spaceSliceOccurrence}

The participant in this SpaceShotOf Link that is the `spaceSliceOccurrence`.

spaceShottedOccurrence : Occurrence {redefines spaceSlicedOccurrence}

The participant in this SpaceShotOf Link that is the `spaceSliced Occurrence`.

Constraints

None.

8.4.2.15 SpaceSliceOf

Element

Association

Description

*SpaceSliceOf*is a *PortionOf* that links its `spaceSliceOccurrence` to its `spaceSlicedOccurrence`, indicating the `spaceSliceOccurrence` extends for exactly the same time and some or all the space of the `spaceSlicedOccurrence` and that the `spaceSliceOccurrence` is of the same of lower `innerSpaceDimension` than the `spaceSliceOccurrence`. This means every *Occurrence/ is a SpaceSliceOf itself and SpaceSliceOf is transitive.*

General Types

PortionOf

Features

spaceSlicedOccurrence : Occurrence {redefines portionedOccurrence}

The participant in this SpaceSliceOf Link that is the `portionedOccurrence`.

spaceSliceOccurrence : Occurrence {redefines portionOccurrence}

The participant in this SpaceSliceOf Link that is the `portionOccurrence`.

Constraints

None.

8.4.2.16 TimeSliceOf

Element

Association

Description

*TimeSliceOf*is a *PortionOf* that links its `timeSliceOccurrence` to its `timeSlicedOccurrence`, indicating that extend for exactly the same time and some or all the space of this Occurrence, including itself. This means every *Occurrence/ is a PortionOf itself.*

General Types

PortionOf

Features

timeSlicedOccurrence : Occurrence {redefines portionedOccurrence}

The participant in this TimeSliceOf Link that is the `portionedOccurrence`.

timeSliceOccurrence : Occurrence {redefines portionOccurrence}

The participant in this TimeSliceOf Link that is the `portionOccurrence`.

Constraints

None.

8.4.2.17 Within

Element

Association

Description

Within classifies all and only links that are *HappensDuring* and *InsideOf*. They link their `largerOccurrence` to their `smallerOccurrence`, indicating the `largerOccurrence` completely overlaps the `smallerOccurrence` in time and space (all four dimensional points of the `smallerOccurrence` *HappensDuring* and are *InsideOf* the `largerOccurrence`). This means every *Occurrence is Within itself and Within is transitive.*

General Types

HappensDuring
InsideOf

Features

largerOccurrence : Occurrence {redefines largerSpace, longerOccurrence}

The participant in this Within Link that is the `longerOccurrence` and `largerSpace`.

smallerOccurrence : Occurrence {redefines shorterOccurrence, smallerSpace}

The participant in this Within Link that is the `shorterOccurrence` and `smallerSpace`.

Constraints

None.

8.4.2.18 WithinBoth

Element

Association

Description

WithinEachOther is a Within and its inverse. This means the linked Occurrences completely overlap each other in space and time (they occupy the same four dimensional region). This means every *Occurrence/ is WithinEachOther with itself and WithinEachOther is transitive.*

General Types

Within

Features

thatOccurrence : Occurrence {redefines largerOccurrence}

thisOccurrence : Occurrence {redefines smallerOccurrence}

Constraints

None.

8.4.2.19 Without

Element

Association

Description

Without classifies all links that are *HappensDuring* or *InsideOf*, or both. They link their `separateOccurrenceToo` to their `separateOccurrence`, indicating that the *Occurrences* do not overlap in time or space (no four dimensional point is in both *Occurrences*). This means no *Occurrence/ is Without itself.*

General Types

BinaryLink

Features

separateOccurrence : Occurrence {redefines target}

The second participant in this Without Link.

separateOccurrenceToo : Occurrence {redefines source}

The first participant in this Without Link.

Constraints

None.

8.5 Objects

8.5.1 Objects Overview

Objects are *Occurrences* that take up a single region of time and space, even though they might be in multiple places over time. *Object* is the most general Structure, while *`objects`* is the most general Feature typed by Structures (see 7.4.3 and compare to *Performances* in 8.6.1). *Objects* and *Performances* do not overlap, but *Performances* can Involve *Objects*, which can *Perform Performances* (see 8.6.1).

LinkObjects are *Objects* that are also *Links*, and *`linkObjects`* is the most general Feature typed by *LinkObject*. *LinkObjects* occupy time and space, like other *Objects*, with potentially varying relationships to other things over time, except for which things are its *`participants`* (the things being linked), identified by its `associationEnd` Features (the "ends" of a link are permanent, though *`participants`* can be *Occurrences* with changing relationships to other things). The values of *LinkObject* Features that are not `associationEnds` can change over time. *LinkObjects* can exist between the same *Occurrences* for only some of the time those *Occurrences* exist, reflecting changing relationships of those *Occurrences*. *BinaryLinkObjects* are *BinaryLinks* that are also *LinkObjects*, and *`binaryLinkObjects`* is the most general Feature typed by *BinaryLinkObject*.

8.5.2 Elements

8.5.2.1 BinaryLinkObject

Element

AssociationStructure

Description

General Types

LinkObject
BinaryLink

Features

source : Anything [0..*]

target : Anything [0..*]

Constraints

None.

8.5.2.2 binaryLinkObjects

Element

Feature

Description

General Types

linkObjects
BinaryLinkObject
binaryLinks

Features

[no name] : Anything

[no name] : Anything

Constraints

None.

8.5.2.3 Body

Element

Structure

Description

Objects of `innerSpaceDimension` 3.

General Types

Object

Features

innerSpaceDimension : Integer {redefines innerSpaceDimension}

volume

Constraints

None.

8.5.2.4 Curve

Element

Structure

Description

Objects of `innerSpaceDimension` 1.

General Types

Object

Features

innerSpaceDimension : Integer {redefines innerSpaceDimension}

Constraints

None.

8.5.2.5 LinkObject

Element

AssociationStructure

Description

LinkObject is the most general AssociationStructure (M1 instance of M2 AssociationStructure). All other AssociationStructures (in libraries or user models) specialize it (directly or indirectly).

General Types

Object
Link

Features

None.

Constraints

None.

8.5.2.6 linkObjects

Element

Feature

Description

`linkObjects` is a specialization of `links` and `objects` restricted to type LinkObject. It is the most general feature typed by LinkObject. All other Features typed by LinkObject or its specializations (in libraries or user models) specialize it (directly or indirectly).

General Types

LinkObject
links
objects

Features

None.

Constraints

None.

8.5.2.7 Object

Element

Structure

Description

An Object is an Occurrence that is not a Performance. It is most general Structure (M1 instance of M2 Structure). All other Structures (in libraries or user models) specialize it (directly or indirectly).

General Types

Occurrence

Features

enactedPerformance : Performance [0..*] {subsets timeEnclosedOccurrences}

Performances that are enacted by this object.

involvedPerformance : Performance [0..*]

Performances in which this Object is involved.

structuredSpaceBoundary : StructuredSpaceObject [0..1] {subsets spaceBoundary}

A `spaceBoundary` that is a StructuredSpaceObject.

Constraints

None.

8.5.2.8 objects

Element

Feature

Description

`objects` is a specialization of `occurrences` restricted to type Object. It is the most general feature typed by Object. All other Features typed by Object or its specializations (in libraries or user models) specialize it (directly or indirectly).

General Types

occurrences
Object

Features

None.

Constraints

None.

8.5.2.9 Point

Element

Structure

Description

Objects of `innerSpaceDimension` 0.

General Types

Object

Features

innerSpaceDimension : Integer {redefines innerSpaceDimension}

Constraints

None.

8.5.2.10 StructuredSpaceObject

Element

Structure

Description

Objects that are broken up into smaller `structuredSpaceCells` of the same or lower `innerSpaceDimension`: `faces` of `innerSpaceDimension` 2, `edges` of `innerSpaceDimension` 1, and `vertices` of `innerSpaceDimension` 0, with the highest of these being the `innerSpaceDimension` of the StructuredSpaceObject. Boundaries of `structuredSpaceObjectCells` are the union of others of lower `innerSpaceDimension` (edges and vertices on the boundary of faces, and vertices on the boundary of edges), some of which overlap when this StructuredSpaceObject `isClosed` (`faces` overlap at their `edges` and/or `vertices`, while `edges` overlap at their `vertices`), as required to be a `spaceBoundary` of an Object .

General Types

Object

Features

cellOrientation : Integer [0..1]

A nested feature of `structuredSpaceObjectCell` that gives them a "direction" (1 or -1) or none (0). For example, the `cellOrientation` of a `face` indicates to which side the "positive" normal vector points, of an `edge` the positive direction along the edge, and of a `vertex` the positive direction "in or out" of it. When the `cellOrientation` of all `edges` and `vertices` are given, and the StructuredSpaceObject `isClosed`, the `cellOrientations` of the (completely) overlapping ones sum to zero.

edges : Curve [0..*] {subsets structuredSpaceObjectCells, ordered}

The `structuredSpaceObjectCells` of `innerSpaceDimension` 1 in this StructuredSpaceObject.

faces : Surface [0..*] {subsets structuredSpaceObjectCells, ordered}

The `structuredSpaceObjectCells` of `innerSpaceDimension` 2 in this StructuredSpaceObject.

/innerSpaceDimension : Integer {redefines innerSpaceDimension}

Highest `innerSpaceDimension` of the `structuredSpaceObjectCells`.

structuredSpaceObjectCells : StructuredSpaceObject [1..*] {subsets spaceSlices}

All and only the `spaceSlices` of this StructuredSpaceObject that are its `faces`, `edges`, and `vertices`.

vertices : Point [0..*] {subsets structuredSpaceObjectCells, ordered}

The `structuredSpaceObjectCells` of `innerSpaceDimension` 0 in this StructuredSpaceObject.

Constraints

None.

8.5.2.11 Surface

Element

Structure

Description

Objects of `innerSpaceDimension` 2.

General Types

Object

Features

genus : Integer [0..1]

The number of "holes" in this Surface, assuming it `isClosed`. For example, it is 0 for spheres and 1 for toruses, including one-handled coffee cups.

innerSpaceDimension : Integer {redefines innerSpaceDimension}

Constraints

None.

8.6 Performances

8.6.1 Performances Overview

Performances

Performances are *Occurrences* that can be spread out in disconnected portions of space and time. *Performance* is the most general Behavior, while *`performances`* is the most general Feature typed by Behaviors (see 7.4.6.1 and compare to *Objects* in 8.5.1). *Performances* can coordinate others that *HappenDuring* them, identified as their *`subperformances`* (see Steps in 7.4.6.1). *Performances* also coordinate and potentially affect other things, some of which might come into existence (start, be "created") or cease of exist (end, be "destroyed") during a Performance, and some that might be used without being affected at all ("catalysts"). Some might be *Objects*, identified as a *Performance*'s *`involvedObjects`*, some of which might be "responsible" for (enact, *Perform*) a *Performance*, identified as its *`performers`*. *Performances* can also accept things as input or provide them as output (see Parameters paragraph in 7.4.6.1).

Evaluations

Evaluations are *Performances* that produce at most one thing (value) identified by their `result` parameter. *Evaluation* is the most general Function, while *`evaluations`* is the most general Feature identifying them, typed by Functions (see 7.4.7.1). In other respects *Evaluations* are like any other *Performance*.

LiteralEvaluations are *Evaluations* with exactly one `result`, specified as a constant in a model via classification by LiteralExpression (see 7.4.8.1 for this and the rest of the paragraph). *LiteralEvaluation* is the most general LiteralExpression, specialized in the same way, and `literalEvaluations` is the most general feature identifying them, also similarly specialized.

BooleanEvaluations are *Evaluations* (but not *LiteralEvaluations*) with exactly one `true` or `false` `result`. *BooleanEvaluation* is the most general Predicate, and `booleanEvaluations` is the most general feature identifying them, specialized (incompletely) into those that always have `true` or always `false` `results`, `trueEvaluations` and `falseEvaluations`, respectively. *LiteralBooleanEvaluation*s are *LiteralEvaluations* and *BooleanEvaluations*, with `result` specified in a model, potentially identified by `trueEvaluations` or `falseEvaluations`, or one of their specializations.

NullEvaluations are *Evaluations* that produce no values for their `result`. *NullEvaluation* is the most general NullExpression, and `nullEvalutions` is the most general Feature typed by NullExpression (see 7.4.8.1).

8.6.2 Elements

8.6.2.1 BooleanEvaluation

Element

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 Types

Evaluation

Features

result : Boolean {redefines result}

The Boolean result of this BooleanExpression.

Constraints

None.

8.6.2.2 booleanEvaluations

Element

BooleanExpression

Description

`booleanEvaluations` is a specialization of `evaluations` restricted to type BooleanEvaluation.

General Types

BooleanEvaluation

evaluations

Features

None.

Constraints

None.

8.6.2.3 Evaluation

Element

Function

Description

An Evaluation is a Performance that ends with the production of a result.

General Types

Performance

Features

result : Anything [0..*] {nonunique}

The `result` is the outcome of the Evaluation.

Constraints

None.

8.6.2.4 evaluations

Element

Expression

Description

`evaluations` is a specialization of `performances` for Evaluations of functions.

General Types

performances
Evaluation

Features

None.

Constraints

None.

8.6.2.5 falseEvaluations

Element

BooleanExpression

Description

`booleanEvaluations` is a specialization of `evaluations` restricted to type BooleanEvaluation.

General Types

booleanEvaluations

Features

[no name] : LiteralEvaluation

Constraints

None.

8.6.2.6 Involves

Element

Association

Description

Involves classifies relationships between Performances and Objects.

General Types

None.

Features

None.

Constraints

None.

8.6.2.7 LiteralEvaluation

Element

Function

Description

LiteralEvaluation is a specialization of Evaluation for the case of LiteralExpressions.

General Types

Evaluation

Features

result : DataValue {redefines result}

The result of this LiteralEvaluation, which is always a single DataValue.

Constraints

None.

8.6.2.8 literalEvaluations

Element

Expression

Description

`literalEvaluations` is a specialization of `evaluations` restricted to type LiteralEvaluation.

General Types

LiteralEvaluation
evaluations

Features

None.

Constraints

None.

8.6.2.9 NullEvaluation

Element

Function

Description

NullEvaluation is a specialization of Evaluation for the case of null expressions.

General Types

Evaluation

Features

result : Anything {redefines result}

The result of this NullEvaluation, which always must be empty (i.e., "null").

Constraints

None.

8.6.2.10 nullEvaluations

Element

Expression

Description

`evaluations` is a specialization of `performances` for Evaluations of functions.

General Types

NullEvaluation
evaluations

Features

None.

Constraints

None.

8.6.2.11 Performance

Element

Behavior

Description

A Performance is an Occurrence that applies constraints to how Objects interact or change over its life.

General Types

Occurrence

Features

involvedObject : Object [0..*]

Objects that are involved in this Performance.

performer : Object [0..*]

Objects that enact this performance.

subperformances : Performance [0..*] {subsets timeEnclosedOccurrences}

Constraints

None.

8.6.2.12 performances

Element

Step

Description

`performances` is the most general feature for Performances of behaviors.

General Types

Performance
things

Features

None.

Constraints

None.

8.6.2.13 Performs

Element

Association

Description

Performs is a specialization of Involves that asserts that the `performer` enacts the behavior carried out by the `enactedPerformance`.

General Types

Involves

Features

None.

Constraints

None.

8.6.2.14 trueEvaluations

Element

BooleanExpression

Description

`booleanEvaluations` is a specialization of `evaluations` restricted to type BooleanEvaluation.

General Types

booleanEvaluations

Features

[no name] : LiteralEvaluation

Constraints

None.

8.7 Transfers

8.7.1 Transfers Overview

Transfers are *Performances* that are also *BinaryLinks*, defined to ensure the things provided by their *`source`* *Occurrence* (via output Features) are accepted by their *`target`* *Occurrence* (via input Features, see Feature Direction in 7.3.2.1). They do this by specifying the existence of *Links* between their *`source`* / *`target`* *Occurrence* and values of the output / input Features of those *Occurrences*, as identified by *`sourceOutputLink`* and *`targetOutputLink`*, respectively. These two Connectors are typed by *BinaryLink*, and can be redefined to more specialized associations when *Transfer* is reused in models. The outputs of the *`source`* *Occurrence* (the things being "transferred") are identified as the *`transferPayload`* of *`sourceOutputLinks`* at the time a *Transfer* starts, also identified as the *`sourceOutput`* of the *Transfer* *`source`*, and as the *Transfer* *`items`*. The inputs of the *`target`* *Occurrence* (the things being "dropped of") are identified as the *`transferPayload`* of *`targetInputLinks`* at the time a *Transfer* ends, also identified as the *`targetOutput`* of the *Transfer* `em>target`, and as the *Transfer* *`items`*. Which things are being transferred does not change during a *Transfer*.

Three Boolean Features of *Transfers* affect their timing and of their *`sourceOutputLinks`* and *`targetOutputLinks`*:

- *`isMove`*: When true, the *`sourceOutputLinks`* end (cease to exist) when the *Transfer* starts, otherwise the *Transfer* has no effect on the *`sourceOutputLinks`*.
- *`isPush`*: When true, the *Transfer* starts when its *`sourceOutputLinks`* do (begin to exist), otherwise the *Transfer* can start anytime after the *`sourceOutputLinks`* do.
- *`isInstant`*: When true, the *Transfer* takes zero time (its *`startShot`* and *`endShot`* are the same, see Portions and Time Slices in 8.4.1).

Transfer and its specializations are binary Interactions, while *`transfers`* is the most general Feature typed by *Transfer* or its specializations, and the most general ItemFlow (see 7.4.9.1). *Transfer* is not the most general binary Interaction, and *`transfers`* is not the most general feature typed by binary Interactions, because binary Interactions can specify more than one *Transfer*.

ItemFlow `sourceOutputFeatures` and `targetInputFeatures` specify which Features of its connected Feature *Occurrences* identify outputs and inputs, respectively (most generally *`sourceOutput`* and *`targetInput`* above, respectively), as well as the kind of outputs and inputs, as its *`itemType`* (most generally the type of *`item`*, above).

8.7.2 Elements

8.7.2.1 Transfer

Element

Interaction

Description

General Types

Performance
BinaryLink

Features

isInstant : Boolean

isMove : Boolean

isPush : Boolean

item : Anything [1..*]

self : Transfer {redefines self}

source : Occurrence [0..*] {subsets toSources}

sourceOutputLink : BinaryLinkObject [1..*]

sourceParticipant : Occurrence {redefines source}

sourceSendShot : Occurrence

target : Occurrence [0..*] {subsets toTargets}

Occurrences whose input is the target of a Transfer of items from this Occurrence.

targetInputLink : BinaryLinkObject [1..*]

targetParticipant : Occurrence {redefines target}

targetReceiveShot : Occurrence

Constraints

None.

8.7.2.2 TransferBefore

Element

Interaction

Description

General Types

Transfer
HappensBefore

Features

source : Occurrence [0..*] {redefines predecessors, source}

sourceParticipant : Occurrence {redefines earlierOccurrence, sourceParticipant}

target : Occurrence [0..*] {redefines target, successors}

Occurrences whose input is the target of a TransferBefore of items from this Occurrence.

targetParticipant : Occurrence {redefines laterOccurrence, targetParticipant}

Constraints

None.

8.7.2.3 transfers

Element

Feature

Description

General Types

Transfer

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

8.7.2.4 transfersBefore

Element

Feature

Description

General Types

TransferBefore
transfers

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

8.8 Feature Referencing Performances

The *FeatureAccessPerfromances* package defines Behaviors used to read and write values of a referenced Feature of an Occurrence as of the time the Performance of the Behavior ends.

8.8.1 Feature Referencing Performances Overview

8.8.2 Elements

8.8.2.1 BooleanEvaluationResultMonitorPerformance

Element

Description

A BooleanEvaluationResultMonitorPerformance is a EvaluationResultMonitorPerformance that waits for changes in the `result` of a BooleanEvaluation identified by `onOccurrence`.

General Types

EvaluationResultMonitorPerformance

Features

afterValues : Boolean {redefines afterValues}

beforeValues : Boolean {redefines beforeValues}

monitoredOccurrence : BooleanEvaluation {subsets timeSlices, redefines monitoredOccurrence}

A `timeSlice` of `onOccurrence` during which its values for `result` change.

onOccurrence : BooleanEvaluation {redefines onOccurrence}

The BooleanEvaluation being monitored for changes in its `result` values.

result : Boolean {redefines result, nonunique}

Redefines BooleanEvaluation::`result` and`monitoredFeature`.

Constraints

None.

8.8.2.2 BooleanEvaluationResultToMonitorPerformance

Element

Description

A BooleanEvaluationResultToMonitorPerformance is a FeatureReferencingPerformance that waits for the `result` of a BooleanEvaluation (identified by `onOccurrence`) to change to either true or false, as indicated by `isToTrue` (defaulting to true). If the `result` is already true (or false), the performance waits for the `result` to become false (or true) before waiting again for it to change back.

General Types

FeatureReferencingPerformance

Features

afterValues : Boolean {redefines values, nonunique}

The values of `monitoredFeature` for `onOccurrence` immediately after they change. Always the same as `isToTrue`.

endWhen : HappensJustBefore

See FeatureMonitorPerformance::`endWhen`. It is restricted to HappensJustBefore in `monitor1` and `monitor2`.

isToTrue : Boolean

monitor1 : BooleanEvaluationResultMonitorPerformance

Waits for the `result` of `onOccurrence` to change.

monitor2 : BooleanEvaluationResultMonitorPerformance [0..1]

Waits for the `result` of `onOccurrence` to change again, only if the change detected by `monitor1` was not the same as `isToTrue`.

onOccurrence : BooleanEvaluation {redefines onOccurrence}

The BooleanEvaluation being monitored for changes in its `result` values.

Constraints

bertmpMonitor1ElseMonitor2

[no documentation]

```
isEmpty(monitor2) == (monitor1.afterValues == isToTrue)
```

8.8.2.3 EvaluationResultMonitorPerformance

Element

Behavior

Description

An EvaluationResultMonitorPerformance is a FeatureMonitorPerformance that waits for changes in `result` of an Evaluation identified by `onOccurrence`. The Predicate being evaluated must be able to produce multiple `results` over time, for example by only using Binding (SelfLink) Connectors between Steps, rather than Successions or ItemFlows, including in its Step `behaviors`.

General Types

FeatureMonitorPerformance

Features

monitoredOccurrence : Evaluation {subsets timeSlices, redefines monitoredOccurrence}

A `timeSlice` of `onOccurrence` during which its values for `result` change.

onOccurrence : Evaluation {redefines onOccurrence}

The Evaluation being monitored for changes in its `result` values.

result : Anything [0..*] {redefines monitoredFeature, nonunique}

Redefines Evaluation::`result` and `monitoredFeature`.

Constraints

None.

8.8.2.4 FeatureAccessPerformance

Element

Behavior

Description

A FeatureAccessPerformance is a FeatureReferencingPerformance where `values` are all the values of `accessedFeature` for `onOccurrence` at the time the Performance ends. Specializations or usages of this narrow `accessedFeature` to particular features.

General Types

FeatureReferencingPerformance

Features

accessedFeature : Anything [0..*] {nonunique}

Feature of `onOccurrence` that has `values` at the time this FeatureAccessPerformance ends.

startingAt : Occurrence {subsets timeSlices}

A `timeslice` of `onOccurrence` that starts when this FeatureAccessPerformance ends.

Constraints

None.

8.8.2.5 FeatureMonitorPerformance

Element

Behavior

Description

A FeatureMonitorPerformance is a FeatureReferencingPerformance that waits for values of `monitoredFeature` to change on `onOccurrence` from what they were when the performance started. The values before and after the change are given by `beforeValues` and `afterValues`

General Types

FeatureReferencingPerformance

Features

afterSnapshot : Occurrence {subsets snapshot}

A `snapShot` of `monitoredOccurrence` just after its values for `monitoredFeature` change.

afterValues : Anything [0..*] {redefines values}

The values of `monitoredFeature` for `monitoredOccurrence` immediately after they change

beforeTimeSlice : Occurrence {subsets timeSlices}

A `timeSlice` of `monitoredOccurrence`, starting at the same time, and ending just before its values for `monitoredFeature` change.

beforeValues : Anything [0..*]

The values of `monitoredFeature` for `monitoredOccurrence` before any change

endWhen : HappensBefore

Succession (Connector typed by HappensBefore) from `afterSnapshot` to the `endShot` of this FeatureMonitorPerformance. Can be specialized to specify how soon the performance should end after the change in `monitoredFeature`.

monitoredFeature : Anything [0..*] {nonunique}

The Feature being monitored for changes in values on `monitoredOccurrence`.

monitoredOccurrence : Occurrence {subsets timeSlices}

A `timeSlice` of `onOccurrence`, starting when this FeatureMonitorPerformance starts, during which the values of `monitoredFeature` change.

Constraints

fmpBeforeAfterValuesNotSame

[no documentation]

```
not beforeValues == afterValues
```

8.8.2.6 FeatureReadEvaluation

Element

Function

Description

A FeatureReadEvaluation is a FeatureAccessPerformance that is a Function providing as its result the values of `accessedFeature` of `onOccurrence` at the time the evaluation ends.

General Types

Evaluation
FeatureAccessPerformance

Features

result : Anything [0..*] {redefines result, values, nonunique}

Values of the Feature being accessed, as an `out` parameter.

Constraints

None.

8.8.2.7 FeatureReferencingPerformance

Element

Behavior

Description

A FeatureReferencingPerformance is a Performance generalizing other Behaviors relating to `values` of a Feature of `onOccurrence`, as specified in the specialized Behaviors.

General Types

Performance

Features

onOccurrence : Occurrence

An Occurrence that has `values` for a Feature determined in specializations of this behavior.

values : Anything [0..*] {nonunique}

Values of a Feature of `onOccurrence`, determined in specializations of this Behavior.

Constraints

None.

8.8.2.8 FeatureWritePerformance

Element

Behavior

Description

A FeatureWritePerformance is a FeatureAccessPerformance that ensures the values of of `onOcurrence` are exactly the `replacementValues` at the time the performance ends.

General Types

FeatureAccessPerformance

Features

replacementValues : Anything [0..*] {redefines values, nonunique}

Values of the Feature being accessed, as an `inout` parameter to replace all the values.

Constraints

None.

8.9 Control Performances

8.9.1 Control Performances Overview

The *ControlPerformances* package defines Behaviors to be used to type Steps that control the sequencing of performance of other Steps, including the following.

DecisionPerformances are *Performances* used by ("decision") Steps to ensure that each *DecisionPerformance* (value) of the Step is the *`earlierOccurrence`* of exactly one *HappensBefore* link of the Successions going out of the Step. Successions going out of `steps` typed by *DecisionPerformance* or its specializations must:

- have connector end multiplicities of 1 towards the Step, and 0..1 away from it.
- subset a Feature of its `featuringBehavior` derived as a chain of the Step and *`DecisionPerformance::outgingHBLink`* (see Feature Chaining in 7.3.4.1).

MergePerformances are *Performances* used by ("merge") Steps to ensure that each *MergePerformance* (value) of the Step is the *`laterOccurrence`* of exactly one *HappensBefore* link of the Successions coming into the step. Successions coming into `steps` typed by MergePerformance or its specializations must:

- have connector end multiplicities of 1 towards the Step, and 0..1 away from it.

- subset a Feature of its `featuringBehavior` derived as a chain of the Step and *`MergePerformance::incomingHBLink`*.

IfPerformances are *Performances* that determine whether one or more clauses occur based on the value of a Boolean argument. The concrete specializations of *IfPerformance* are *IfThenPerformance, IfElsePerformance* and *IfThenElsePerformance.*

LoopPerformances are *Performances* that whose body occurs iteratively as determined by Boolean "while" and "until" conditions.

8.9.2 Elements

8.9.2.1 DecisionPerformance

Element

Behavior

Description

A DecisionPerformance is a Performance that represents the selection of one of the Successions that have the DecisionPerforance behavior as their source. All such Successions must subset the `outgoingHBLink` feature of the source DecisionPerformance. For each instance of DecisionPerformance, the `outgoingHBLink` is an instance of exactly one of the Successions, ordering the DecisionPerformance as happening before an instance of the target of that Succession.

General Types

Performance

Features

outgoingHBLink : HappensBefore

Constraints

None.

8.9.2.2 IfElsePerformance

Element

Behavior

Description

An IfElsePerformance is an IfPerformance where `else` occurs after and only after the `ifTest` Evaluation `result` is not true.

General Types

IfPerformance

Features

elseClause : Occurrence [0..1]

Constraints

None.

8.9.2.3 IfPerformance

Element

Behavior

Description

An IfPerformance is a Performance that determines whether the `if` Evaluation `result` is true (by whether the `ifTrue` connector has a value).

General Types

Performance

Features

ifTest : BooleanEvaluation

trueLiteral : LiteralEvaluation

Constraints

None.

8.9.2.4 IfThenElsePerformance

Element

Behavior

Description

An IfThenElsePerformance is an IfThenPerformance and an IfElsePerformance.

General Types

IfElsePerformance
IfThenPerformance

Features

None.

Constraints

None.

8.9.2.5 IfThenPerformance

Element

Behavior

Description

An IfThenPerformance is an IfPerformance where `then` occurs after and only after the `if` Evaluation `result` is true.

General Types

IfPerformance

Features

thenClause : Occurrence [0..1]

Constraints

None.

8.9.2.6 LoopPerformance

Element

Behavior

Description

A LoopPerformance is a Performance where `body` occurs repeatedly in sequence (iterates) as long as the `while` evaluation result is true before each iteration (and after the previous one, except the first time) and the `until` evaluation result is not true after each iteration and before the next one (except the last one).

General Types

Performance

Features

body : Occurrence [0..*]

untilDecision : IfElsePerformance [0..*]

untilTest : BooleanEvaluation [0..*]

whileDecision : IfThenPerformance [1..*]

whileTest : BooleanEvaluation [1..*]

Constraints

None.

8.9.2.7 MergePerformance

Element

Behavior

Description

A MergePerformance is a Performance that represents the merging of all Successions that target the MergePerforance behavior. All such Successions must subset the `incomingHBLink` feature of the target MergePerformance. For each instance of MergePerformance, the `incomingHBLink` is an instance of exactly one of the Successions, ordering the MergePerformance as happening after an instance of the source of that Succession.

General Types

Performance

Features

incomingHBLink : HappensBefore

Constraints

None.

8.10 Transition Performances

8.10.1 Transition Performances Overview

The *TransitionPerformances* package contains a library model of the semantics of conditional transitions between *Occurrences*, including the performance of specified Behaviors when the transition occurs.

TransitionPerformances are *Performances* used to

- determine whether a Succession going out of an Occurrence Feature (`Succession::sourceFeature`) has values (*HappensBefore* links), based on *Occurrences* of `sourceFeature` and other conditions, including ending of *Transfers*.
- perform specified Behaviors for each value of the Succession above.

The Succession constrained by a *TransitionPerformance* is specified by a Connector between the Succession and its `transitionStep` (see Successions in 7.4.5.1), a unique Step typed by *TransitionPerformance* or a specialization of it, of the same Behavior as the Succession. This connector is

- typed by an Association defined to give a value to the *transitionLink* of *TransitionPerformances*,
- has connector end multiplicity 0.1 on the Succession end and 1 on the *TransitionPerformance* Step end.

The connector end multiplicities above ensure every *HappensBefore* link of the Succession is paired with a unique *TransitionPerformance* that has its conditions satisfied for that *Link*, while all the other *TransitionPerformances* of `transitionStep` fail their conditions and have no values for *`transitionLink`*.

The `transitionStep` above is also connected to the Succession's `sourceFeature`, because conditions on the Succession depend on each *Occurrence* of its `sourceFeature` separately, which *TransitionPerformances* identify as their *`transitionLinkSource`*. This connector is

- typed by an Association defined to give a value to the `transitionLinkSource` of *TransitionPerformances*.
- with connector end multiplicity 1 on both ends.

The connector end multiplicities above ensure every *Occurrence* of the Succession's `sourceFeature` is paired with a unique *TransitionPerformance*, and vice-versa, that determines whether the Succession has a value (*HappensBefore* link) for that *Occurrence*.

TransitionPerformances with a `transitionLink` must satisfy these conditions:

- all *Transfers* identified by `trigger` must happen before all *Evaluations* identified by guard.
- all *Evaluations* identified by `guard` must have `result` value `true`.

The `effect` of a *TransitionPerformance* can have values (*Performances*) only if the above conditions hold. The `effect` *Performances* must happen after the `guards` and before the `laterOccurrence` of `transitionLink`.

Usages of (Steps typed by) *TransitionPerformance* or its specializations can redefine or subset `guard` and `effect` to specify how they are carried out, as well as specify how `triggers` are identified. These usages can

- be `steps` of any Behavior (not only "state machines"), as well as constrain Successions going out of any kind of Step (not only those identifying *StatePerformances*, see 8.11.1).
- employ any method of identifying `triggers`, including requiring none at all, as well as constraining *Transfer* `targets` to be, for example, the *StatePerformance* itself, or a *Performance* it is a `subperformance` of, or an *Object* enacting that *Performance*.

TransitionPerformances are either *StateTransitionPerformances* or *NonStateTransitionPerformances*, depending on whether the `transitionLinkSource` is a *StatePerformance* or not. Both ensure `guards` happen before the `laterOccurrence` of `transitionLink`, in case there are no `effects`, but do this in different ways (see 8.11.1 about StateTransitionPerformances).

8.10.2 Elements

8.10.2.1 NonStateTransitionPerformance

Element

Behavior

Description

General Types

TransitionPerformance

Features

None.

Constraints

None.

8.10.2.2 TPCGuardConstraint

Element

Association

Description

General Types

BinaryLink

Features

constrainedGuard : Evaluation {redefines target}

constrainedHBLink : HappensBefore {redefines source}

guardedBy : Evaluation [0..*] {redefines toTargets}

guards : HappensBefore [0..1] {redefines toSources}

true : Boolean

Constraints

None.

8.10.2.3 TransitionPerformance

Element

Behavior

Description

General Types

Performance

Features

effect : Performance [0..*]

guard : Evaluation [0..*]

guardConstraint : TPCGuardConstraint [0..*]

transitionLink : HappensBefore [0..1]

transitionLinkSource : Performance

trigger : Transfer [0..*]

Constraints

None.

8.11 State Performances

8.11.1 State Performances Overview

The *StatePerformance* package contains a library model of the semantics of state-based behavior, including *StatePerformances* and *StateTransitionPerformances* between them.

StatePerformances are *DecisionPerformances* (see 8.9.1) that

- only have Steps defined in this library, or specialized from them.
- can identify *Transfers* that happen before the last *Performance* of the above Steps (see `exit` below).

Usages of *StatePerformance* can specialize the library Steps to specify how they are carried out, as well as specify how the *Transfer* above is identified. Additional modeler-defined Steps must subset the `middle` of the library Steps:

- `entry` [1]: happens before all *Performances* of `middle`.
- `middle` [1..*]: happen during the *Performance* of `do`.
- `do` [1]: one of the *Performances* of `middle` that starts before the others and ends after them.
- `exit` [1]: happens after all *Performances* of `middle` and the end of the *Transfers* identified by the *StatePerformance* (see `accepted` below).

StatePerformances identify *Transfers* that happen before the *Performance* of `exit`, as specified by usages of *StatePerformance* by redefining these library Steps:

- `acceptable` [*]: candidates for being identified as `accepted`.
- `accepted` [0..1]: one of the `acceptable` transfers. This must have a value if `acceptable` does.

If `isTriggerDuring` is `true` then `accepted` must end during the *StatePerformance*. If `isAcceptFirst` is `true`, `accepted` must end before the other `acceptable` ones.

Steps typed by *StatePerformances* can

- be `steps` of any Behavior (not only "state machines")
- employ any method of identifying *Transfers* needed to start ("trigger") their `exit`, including none at all, as well as requiring their `targets` to be, for example, the *StatePerformance* itself, or a *Performance* it is a `subperformance` of, or an *Object* enacts that *Performance*.
- have outgoing Successions constrained to have values (*HappensBefore* links) or not based on their `earlierOccurrences` (a *StatePerformance* of the step) and other conditions.
- be used in conjunction with other Steps typed by `TransitionPerformances` (see 8.10.1) to determine which Succession going out of a Step "is chosen" (has a *HappenBeforeLink* value with a *StatePerformance* of that Step as its `earlierOccurrence`).

StateTransitionPerformances are *TransitionPerformances* (see 8.10.1) that have a *StatePerformance* as their `transitionLinkSource`. *StateTransitionPerformance*

- `triggers` subset those of its `transitionLinkSource`. If its `isTriggerDuring` is `true`, `triggers` must end during its `transitionLinkSource`.
- `guards` happen after the `middle` Step of its `transitionLinkSource` and before the `exit` Step.

8.11.2 Elements

8.11.2.1 StatePerformance

Element

Behavior

Description

General Types

DecisionPerformance

Features

/acceptable : Transfer [0..*] {union}

accepted : Transfer [0..1] {subsets acceptable}

do : Performance {subsets middle}

entry : Performance {subsets timeEnclosedOccurrences}

exit : Performance {subsets timeEnclosedOccurrences}

isAcceptFirst : Boolean

isTriggerDuring : Boolean

/middle : Performance [1..*] {subsets timeEnclosedOccurrences, union}

Constraints

None.

8.11.2.2 StateTransitionPerformance

Element

Behavior

Description

General Types

TransitionPerformance

Features

isTriggerDuring : Boolean

transitionLinkSource : StatePerformance {redefines transitionLinkSource}

Constraints

None.

8.12 Clocks

8.12.1 Clocks Overview

This package models *Clocks* that provide an advancing numerical reference usable for quantifying the time of an *Occurrence*.

8.12.2 Elements

8.12.2.1 BasicClock

Element

Structure

Description

A *BasicClock* is a *Clock* whose *currentTime* is a *Real* number.

General Types

Clock

Features

currentTime : Real {redefines currentTime}

Constraints

None.

8.12.2.2 BasicDurationOf

Element

Function

Description

BasicDurationOf returns the *DurationOf* an *Occurrence* as a *Real* number relative to a *BasicClock*.

General Types

DurationOf

Features

clock : BasicClock {redefines clock}

duration : Real {redefines duration}

o : Occurrence {redefines o}

Constraints

None.

8.12.2.3 BasicTimeOf

Element

Function

Description

`BasicTimeOf` returns the *`TimeOf`* an *`Occurrence`* as a *`Real`* number relative to a *`BasicClock`*.

General Types

TimeOf

Features

clock : BasicClock {redefines clock}

o : Occurrence {redefines o}

timeValue : Real {redefines timeValue}

Constraints

None.

8.12.2.4 Clock

Element

Structure

Description

A *`Clock`* provides a scalar *`currentTime`* that advances montonically over its lifetime. *`Clock`* is an abstract base Structure that can be specialized for different kinds of time quantification (e.g., discrete time, continuous time, time with units, etc.).

General Types

Object

Features

currentTime : ScalarValue

A numerical time reference that advances over the lifetime of the *`Clock`*.

Constraints

timeFlowConstraint

The *currentTime* of a snapshot of a *Clock* is equal to the *TimeOf* the snapshot relative to that *Clock*.

8.12.2.5 defaultClock

Element

Feature

Description

defaultClock is a single *Clock* that can be used as a default.

General Types

Clock
objects

Features

None.

Constraints

None.

8.12.2.6 DurationOf

Element

Function

Description

DurationOf returns the duration of a given *Occurrence* relative to a given *Clock*, which is equal to the *TimeOf* the end snapshot of the *Occurrence* minus the *TimeOf* its start snapshot.

General Types

Evaluation

Features

clock : Clock

duration : NumericalValue

o : Occurrence

Constraints

None.

8.12.2.7 TimeOf

Element

Function

Description

TimeOf returns a scalar *timeValue* for a given *Occurrence* relative to a given *Clock*. The *timeValue* is the time of the start of the *Occurrence*, which is considered to be synchronized with the snapshot of the *Clock* with a *currentTime*timeValue.

General Types

Evaluation

Features

clock : Clock

o : Occurrence

timeValue : ScalarValue

Constraints

timeContinuityConstraint

If one *Occurrence* happens immediately before another, then the *TimeOf* the end snapshot of the first Occurrence equals the *TimeOf* the second *Occurrence*.

startTimeConstraint

The *TimeOf* an *Occurrence* is equal to the time of its start snapshot.

timeOrderingConstraint

If one *Occurrence* happens before another, then the *TimeOf* the end snapshot of the first *Occurrence* is no greater than the *TimeOf* the second *Occurrence*.

8.13 Observation

8.13.1 Observation Overview

This package models a framework for monitoring *Boolean* conditions and notifying registered observers when they change from false to true.

8.13.2 Elements

8.13.2.1 CancelObservation

Element

Behavior

Description

Cancel all observations of a given *ChangeSignal* for a given *Occurrence*.

General Types

Performance

Features

observer : Occurrence

signal : ChangeSignal

Constraints

None.

8.13.2.2 changeCondition

Element

Expression

Description

General Types

None.

Features

None.

Constraints

None.

8.13.2.3 ChangeMonitor

Element

Structure

Description

A `ChangeMonitor` is a collection of ongoing `ChangeSignal` observations for various observer `Occurrences`. It provides convenient operations for starting and canceling the observations it manages.

General Types

Object

Features

cancelObservation : CancelObservation [0..*]

Cancel all observations of a given `ChangeSignal` for a given `Occurrence`.

observations : ObserveChange [0..*]

startObservation : StartObservation [0..*]

Start an observation of a given *ChangeSignal* for a given *Occurrence*.

Constraints

None.

8.13.2.4 ChangeSignal

Element

Structure

Description

A *ChangeSignal* is a signal to be sent when the *Boolean* result of its *changeCondition* Expression changes from false to true.

General Types

Object

Features

changeMonitor : ChangeMonitor

The *ChangeMonitor* responsible for monitoring the *signalCondition*.

signalCondition : BooleanEvaluation

A BooleanExpression whose result is being monitored.

Constraints

None.

8.13.2.5 defaultMonitor

Element

Feature

Description

defaultMonitor is a single *ChangeMonitor* that can be used as a default.

General Types

ChangeMonitor
objects

Features

None.

Constraints

None.

8.13.2.6 ObserveChange

Element

Behavior

Description

Each *Performance* of *ObserveChange* waits for the result of the *Boolean changeCondition* of a given *ChangeSignal* to change from false to true, and, when it does, sends the *ChangeSignal* to a given observer *Occurrence*.

General Types

Performance

Features

changeObserver : Occurrence

changeSignal : ChangeSignal

transfer : TransferBefore [0..1]

After waiting for the condition change (if necessary), then send *changeSignal* to *changeObserver*.

wait : IfThenPerformance

If the result of the *changeSignal.signalCondition* is false, then wait for it to become true:

```
in ifTest { not changeSignal.signalCondition() }
in thenClause : BooleanEvaluationResultToMonitorPerformance {
    in onOccurrence = changeSignal.signalCondition;
}
```

Constraints

None.

8.13.2.7 StartObservation

Element

Behavior

Description

Start an observation of a given *ChangeSignal* for a given *Occurrence*.

General Types

Performance

Features

observer : Occurrence

signal : ChangeSignal

Constraints

None.

8.14 Triggers

8.14.1 Triggers Overview

This package contains functions that return `ChangeSignals` for triggering when a `Boolean` condition changes from false to true, at a specific time or after a specific time delay.

8.14.2 Elements

8.14.2.1 TimeSignal

Element

Structure

Description

A `TimeSignal` is a `ChangeSignal` whose condition is the `currentTime` of a given `Clock` reaching a specific `signalTime`.

General Types

ChangeSignal

Features

signalClock : Clock

The `Clock` whose `currentTime` is being monitored.

signalCondition : BooleanEvaluation {redefines signalCondition}

The `Boolean` condition of the `currentTime` of the `signalClock` being equal to the `signalTime`.

signalTime : NumericalValue

The time at which the `TimeSignal` should be sent.

Constraints

None.

8.14.2.2 TriggerAfter

Element

Function

Description

TriggerAfter returns a monitored *TimeSignal* to be sent to a *receiver* after a certain time *delay* relative to a given *Clock*.

General Types

Evaluation

Features

clock : Clock

The *Clock* to be used as the reference for the time *delay*. The default is the *Clocks::defaultClock*.

delay : NumericalValue

The time duration, relative to the *clock*, after which the *TimeSignal* is sent.

monitor : ChangeMonitor

The *ChangeMonitor* to be used to monitor the *TimeSignal* condition. The default is the *Observation::defaultMonitor*.

receiver : Occurrence

The *Occurrence* to which the *TimeSignal* is to be sent.

signal : TimeSignal

Constraints

None.

8.14.2.3 TriggerAt

Element

Function

Description

TriggerAt returns a monitored *TimeSignal* to be sent to a *receiver* when the *currentTime* of a given *Clock* reaches a specific *time*.

General Types

Evaluation

Features

clock : Clock

The *Clock* to be used as the reference for the *time*. The default is the *Clocks::defaultClock*.

monitor : ChangeMonitor

The *ChangeMonitor* to be used to monitor the *TimeSignal* condition. The default is the *Observation::defaultMonitor*.

receiver : Occurrence

The *Occurrence* to which the *TimeSignal* is to be sent.

signal : TimeSignal

time : NumericalValue

The time instant, relative to the *clock*, at which the *TimeSignal* should be sent.

Constraints

None.

8.14.2.4 TriggerWhen

Element

Function

Description

TriggerWhen returns a monitored *ChangeSignal* for a given *condition*, to be sent to a given *receiver* when the *condition* occurs.

General Types

Evaluation

Features

condition : BooleanEvaluation

The BooleanExpression to be monitored for changing from false to true.

monitor : ChangeMonitor

The *ChangeMonitor* to be used to monitor the *ChangeSignal* condition. The default is the *Observation::defaultMonitor*.

receiver : Occurrence

The *Occurrence* to which the *ChangeSignal* is to be sent.

signal : ChangeSignal

Constraints

None.

8.15 SpatialFrames

8.15.1 SpatialFrames Overview

This package models spatial frames of reference for quantifying the position of points in a three-dimensional space.

8.15.2 Elements

8.15.2.1 CartesianCurrentDisplacementOf

Element

Function

Description

The `CurrentDisplacementOf` two Points relative to a `CartesianSpatialFrame` is a `CartesianThreeVectorValue`.

General Types

CurrentDisplacementOf

Features

clock : Clock {redefines clock}

displacementVector : CartesianThreeVectorValue {redefines displacementVector}

frame : CartesianSpatialFrame {redefines frame}

point1 : Point {redefines point1}

point2 : Point {redefines point2}

Constraints

None.

8.15.2.2 CartesianCurrentPositionOf

Element

Function

Description

The `CurrentPositionOf` a Point relative to a `CartesianSpatialFrame` is a `CartesianThreeVectorValue`.

General Types

CurrentPositionOf

Features

clock : Clock {redefines clock}

frame : CartesianSpatialFrame {redefines frame}

point : Point {redefines point}

positionVector : CartesianThreeVectorValue {redefines positionVector}

Constraints

None.

8.15.2.3 CartesianDisplacementOf

Element

Function

Description

The `DisplacementOf` two Points relative to a `CartesianSpatialFrame` is a `CartesianThreeVectorValue`.

General Types

DisplacementOf

Features

clock : Clock {redefines clock}

displacementVector : CartesianThreeVectorValue {redefines displacementVector}

frame : CartesianSpatialFrame {redefines frame}

point1 : Point {redefines point1}

point2 : Point {redefines point2}

time : NumericalValue {redefines time}

Constraints

None.

8.15.2.4 CartesianPositionOf

Element

Function

Description

The *PositionOf* a Point relative to a *CartesianSpatialFrame* is a *CartesianThreeVectorValue*.

General Types

PositionOf

Features

clock : Clock {redefines clock}

frame : CartesianSpatialFrame {redefines frame}

point : Point {redefines point}

positionVector : CartesianThreeVectorValue {redefines positionVector}

time : NumericalValue {redefines time}

Constraints

None.

8.15.2.5 CartesianSpatialFrame

Element

Structure

Description

A *CartesianSpatialFrame* is a *SpatialFrame* relative to which all position and displacement vectors can be represented as *CartesianThreeVectorValues*.

General Types

SpatialFrame

Features

None.

Constraints

None.

8.15.2.6 CurrentDisplacementOf

Element

Function

Description

The *CurrentDisplacementOf* two *Points* relative to a *SpatialFrame* and *Clock* is the *DisplacementOf* the *Points* relative to the *SpacialFrame* at the *currentTime* of the *Clock*.

General Types

Evaluation

Features

clock : Clock

displacementVector : ThreeVectorValue

frame : SpatialFrame

point1 : Point

point2 : Point

Constraints

None.

8.15.2.7 CurrentPositionOf

Element

Function

Description

The *CurrentPositionOf* a *Point* relative to a *SpatialFrame* and a *Clock* is the *PositionOf* the *Point* relative to the *SpatialFrame* at the *currentTime* of the *Clock*.

General Types

Evaluation

Features

clock : Clock

frame : SpatialFrame

point : Point

positionVector : ThreeVectorValue

Constraints

None.

8.15.2.8 defaultFrame

Element

Feature

Description

defaultFrame is a fixed *SpatialFrame* used as a universal default.

General Types

SpatialFrame

Features

None.

Constraints

None.

8.15.2.9 DisplacementOf

Element

Function

Description

The *DisplacementOf* two *Points* relative to a *SpatialFrame*, at a specific *time* relative to a given *Clock*, is the *displacementVector* computed as the difference between the *PositionOf* the first *Point* and *PositionOf* the second *Point*, relative to that *SpatialFrame*, at that *time*.

General Types

Evaluation

Features

clock : Clock

displacementVector : ThreeVectorValue

frame : SpatialFrame

point1 : Point

point2 : Point

time : NumericalValue

Constraints

zeroDisplacementConstraint

If either *point1* or *point2* occurs within the other, then the *displacementVector* is the zero vector.

```
(point1.spaceTimeEnclosedOccurrences->includes(point2) or
point2.spaceTimeEnclosedOccurrences->includes(point1)) implies
    isZeroVector(displacementVector)
```

8.15.2.10 PositionOf

Element

Function

Description

The *PositionOf* a *Point* relative to a *SpatialFrame*, at a specific *time* relative to a given *Clock*, as a *positionVector* that is a *ThreeVectorValue*.

General Types

Evaluation

Features

clock : Clock

frame : SpatialFrame

point : Point

positionVector : ThreeVectorValue

time : NumericalValue

Constraints

positionTimePrecondition

The given *point* must exist at the given *time*.

```
TimeOf(point.startShot) <= time and
time <= TimeOf(point.endShot)
```

spacePositionConstraint

The result *positionVector* is equal to the *PositionOf* the *Point spaceShot* of the frame that encloses the given *point*, at the given *time*.

```
(frame.spaceShots as Point)->forAll{in p : Point;
    p.spaceTimeEnclosedOccurrences->includes(point) implies
    positionVector == PositionOf(p, time, frame)
}
```

8.15.2.11 SpatialFrame

Element

Structure

Description

General Types

Body

Features

None.

Constraints

None.

8.16 Metaobjects

8.16.1 Metaobjects Overview

This package defines Metaclasses and Features that are related to the typing of syntactic and semantic metadata.

8.16.2 Elements

8.16.2.1 Metaobject

Element

Metaclass

Description

A *`Metaobject`* contains syntactic or semantic information about one or more *`annotatedElements`*. It is the most general Metaclass. All other Metaclasses must subclassify it directly or indirectly.

General Types

Object

Features

annotatedElement : Element [1..*]

The Elements annotated by this *`Metaobject`*. This is set automatically when a *`Metaobject`* is instantiated as the value of a MetadataFeature.

Constraints

None.

8.16.2.2 metaobjects

Element

Feature

Description

`metaobjects` is a specialization of `objects` restricted to type `Metaobject`. It is the most general MetadataFeature. All other MetadataFeatures must subset it directly or indirectly.

General Types

objects
Metaobject

Features

None.

Constraints

None.

8.16.2.3 SemanticMetadata

Element

Metaclass

Description

SemanticMetadata is *Metadata* `that requires its single` *annotatedType* `to directly or indirectly specialize a` *baseType* `that models the semantics for the` *annotatedType*`.`

General Types

Metaobject

Features

annotatedElement : Type {redefines annotatedElement}

The single `annotatedElement` of this SemanticMetadata, which must be a *Type*.

baseType : Type

The required base *Type* for the *annotatedType*.

Constraints

None.

8.17 KerML

8.17.1 KerML Overview

This package contains a reflective KerML model of the KerML abstract syntax.

Release Note: This model is currently incomplete. It includes all KerML abstract syntax metaclasses, but none of their properties.

8.17.2 Elements

```
metaclass AnnotatingElement :> Element;
metaclass Annotation :> Relationship;
metaclass Comment :> AnnotatingElement;
metaclass Documentation :> Annotation;
metaclass TextualRepresentation :> AnnotatingElement;

metaclass Import :> Relationship;
metaclass Membership :> Relationship;
metaclass Namespace :> Element;

metaclass Type :> Namespace;
metaclass Multiplicity :> Feature;
metaclass FeatureMembership :> Membership, TypeFeaturing;
metaclass Specialization :> Relationship;
metaclass Conjugation :> Relationship;
metaclass Disjoining :> Relationship;

metaclass Classifier :> Type;
metaclass Subclassification :> Specialization;

metaclass Feature :> Type;
metaclass Subsetting :> Specialization;
metaclass Redefinition :> Subsetting;
metaclass FeatureTyping :> Specialization;
metaclass TypeFeaturing :> Relationship;
metaclass FeatureChaining :> Relationship;
metaclass EndFeatureMembersip :> FeatureMembership;

metaclass Class :> Classifier;
metaclass DataType :> Classifier;

metaclass Structure :> Class;

metaclass Association :> Classifier, Relationship;
metaclass AssociationStructure :> Association, Structure;

metaclass Connector :> Feature, Relationship;
metaclass BindingConnector :> Connector;
metaclass Succession :> Connector;

metaclass Behavior :> Class;
metaclass Step :> Feature;
metaclass ParameterMembership :> FeatureMembership;

metaclass Function :> Behavior;
metaclass Predicate :> Function;
metaclass Expression :> Step;
metaclass BooleanExpression :> Expression;
metaclass Invariant :> BooleanExpression;
metaclass ReturnParameterMembership :> ParameterMembership;
metaclass ResultExpressionMembership :> FeatureMembership;

metaclass FeatureReferenceExpression :> Expression;
metaclass InvocationExpression :> Expression;
metaclass LiteralExpression :> Expression;
metaclass LiteralInteger :> LiteralExpression;
metaclass LiteralRational :> LiteralExpression;
```

```
metaclass LiteralBoolean :> LiteralExpression;
metaclass LiteralString :> LiteralExpression;
metaclass LiteralInfinity :> LiteralExpression;
metaclass NullExpression :> Expression;
metaclass OperatorExpression :> InvocationExpression;
metaclass FeatureChainExpression :> OperatorExpression;
metaclass CollectExpression :> OperatorExpression;
metaclass SelectExpression :> OperatorExpression;

metaclass Interaction :> Behavior, Association;
metaclass ItemFlow :> Step, Connector;
metaclass SuccessionItemFlow :> ItemFlow, Succession;

metaclass FeatureValue :> Membership;

metaclass MultiplicityRange :> Multiplicity;

metaclass Metaclass :> Structure;
metaclass MetadataFeature :> AnnotatingElement, Feature;

metaclass Package :> Namespace;
metaclass ElementFilterMembership :> Membership;
```

8.18 Scalar Values

8.18.1 Scalar Values Overview

This package contains a basic set of primitive scalar (non-collection) data types. These include *Boolean* and *String* types and a hierarchy of concrete *Number* types, from the most general type of *Complex* numbers to the most specific type of *Positive* integers.

8.18.2 Elements

8.18.2.1 Boolean

Element

DataType

Description

Boolean is a ScalarValue type whose instances are true and false.

General Types

ScalarValue

Features

None.

Constraints

None.

8.18.2.2 Complex

Element

DataType

Description

Complex is the type of complex numbers.

General Types

Number

Features

None.

Constraints

None.

8.18.2.3 Integer

Element

DataType

Description

Integer is the type of mathematical integers, extended with values for positive and negative infinity.

General Types

Rational

Features

None.

Constraints

None.

8.18.2.4 Natural

Element

DataType

Description

Natural is the type of non-negative integers, extended with a value for positive infinity.

General Types

DataValue
Integer

Features

None.

Constraints

None.

8.18.2.5 Number

Element

DataType

Description

Number is the base type for all NumericalValue types that represent numbers.

General Types

NumericalValue

Features

None.

Constraints

None.

8.18.2.6 NumericalValue

Element

DataType

Description

NumericalValue is the base type for all ScalarValue types that represent numerical values.

General Types

ScalarValue

Features

None.

Constraints

None.

8.18.2.7 Positive

Element

DataType

Description

Positive is the type of positive integers (not including zero), extended with a value for positive infinity.

General Types

Natural

Features

None.

Constraints

None.

8.18.2.8 Rational

Element

DataType

Description

Rational is the type of rational numbers, extended with values for positive and negative infinity.

General Types

Real

Features

None.

Constraints

None.

8.18.2.9 Real

Element

DataType

Description

Real is the type of mathematical (extended) real numbers. This includes both rational and irrational numbers, and values for positive and negative infinity.

General Types

Complex

Features

None.

Constraints

None.

8.18.2.10 ScalarValue

Element

DataType

Description

A ScalarValue is a DataValue whose instances are considered to be primitive, not collections or structures of other values.

General Types

DataValue

Features

None.

Constraints

None.

8.18.2.11 String

Element

DataType

Description

String is a ScalarValue type whose instances are strings of characters.

General Types

ScalarValue

Features

None.

Constraints

None.

8.19 Collections

8.19.1 Collections Overview

This package defines a standard set of *Collection* data types. Unlike sequences of values defined directly using multiplicity, these data types allow for the possibility of collections as elements of collections.

8.19.2 Elements

8.19.2.1 Array

Element

DataType

Description

An Array is a fixed size, multi-dimensional Collection of which the `elements` are nonunique and ordered. Its `dimensions` specify how many dimensions the array has, and how many elements there are in each dimension. The `rank` is equal to the number of `dimensions`. The `flattenedSize` is equal to the total number of `elements` in the array.

Feature `elements` is a flattened sequence of all elements of an Array and can be accessed by a tuple of indices. The number of indices is equal to `rank`. The `elements` are packed according to row-major convention, as in the C programming language.

Note 1. Feature `dimensions` may be empty, which denotes a zero dimensional array, allowing an Array to collapse to a single element. This is useful to allow for specialization of an Array into a type restricted to represent a scalar. The flattenedSize,/code> of a zero dimensional array is 1.

Note 2. An Array can also represent the generalized concept of a mathematical matrix of any rank, i.e. not limited to rank two.

General Types

OrderedCollection

Features

dimensions : Positive [0..*] {ordered, nonunique}

flattenedSize : Positive

rank : Natural

Constraints

sizeConstraint

[no documentation]

```
flattenedSize == size(elements)
```

8.19.2.2 Bag

Element

DataType

Description

A Bag is a variable size Collection of which the `elements` are unordered and nonunique.

General Types

Collection

Features

None.

Constraints

None.

8.19.2.3 Collection

Element

DataType

Description

A Collection is an abstract DataType that represents a collection of elements of a given type.

A Collection is either mutable or immutable, or mutability is unspecified.

TODO: Decide on whether to add Mutability, and if so, how.

General Types

Anything

Features

elements : Anything [0..*] {nonunique}

Constraints

None.

8.19.2.4 KeyValuePair

Element

DataType

Description

A KeyValuePair is an abstract DataType that represents a tuple of a `key` and an associated value `val`.

General Types

DataValue

Features

key : Anything

val : Anything

Constraints

None.

8.19.2.5 List

Element

DataType

Description

A Sequence is a variable size Collection of which the `elements` are nonunique and ordered.

General Types

OrderedCollection

Features

None.

Constraints

None.

8.19.2.6 Map

Element

DataType

Description

A Map is a variable size Collection of which the `elements` are KeyValuePairs. The keys must be unique within in the Map. The values need not be unique.

General Types

UniqueCollection

Features

elements : KeyValuePair [0..*] {redefines elements}

Constraints

None.

8.19.2.7 OrderedCollection

Element

DataType

Description

An OrderedCollection is a Collection of which the `elements` are ordered, and not necessarily unique).

General Types

Collection

Features

elements : Anything [0..*] {redefines elements, ordered, nonunique}

Constraints

None.

8.19.2.8 OrderedMap

Element

DataType

Description

An OrderedMap is a variable size Map that maintains ordering of its elements.

The ordering may be by key of the KeyValuePair elements, or by order of construction, or any other method. The essential aspect is that ordering is maintained and guaranteed across accesses to the OrderedMap.

General Types

Map
OrderedCollection

Features

elements : KeyValuePair [0..*] {redefines elements, ordered}

Constraints

None.

8.19.2.9 OrderedSet

Element

DataType

Description

An OrderedSet is a variable size Collection of which the `elements` are unique and ordered.

General Types

OrderedCollection
UniqueCollection

Features

elements : Anything [0..*] {redefines elements, ordered}

Constraints

None.

8.19.2.10 Set

Element

DataType

Description

A Set is a variable size Collection of which the `elements` are unique and unordered.

General Types

UniqueCollection

Features

None.

Constraints

None.

8.19.2.11 UniqueCollection

Element

DataType

Description

A UniqueCollection is a Collection of which the `elements` are unique, and not necessarily ordered).

General Types

Collection

Features

elements : Anything [0..*] {redefines elements}

Constraints

None.

8.20 Vector Values

8.20.1 Vector Values Overview

8.20.2 Elements

8.20.2.1 CartesianThreeVectorValue

Element

DataType

Description

A `CartesianThreeVectorValue` is a `NumericalVectorValue` that is both Cartesian and has dimension 3.

General Types

ThreeVectorValue
CartesianVectorValue

Features

None.

Constraints

None.

8.20.2.2 CartesianVectorValue

Element

DataType

Description

A `CartesianVectorValue` is a `NumericalVectorValue` for which there are specific implementations in `VectorFunctions` of the abstract vector-space functions.

Note: The restriction of the element type to `Real` is to facilitate the complete definition of these functions.

General Types

NumericalVectorValue

Features

elements : Real [0..*] {redefines elements}

Constraints

None.

8.20.2.3 NumericalVectorValue

Element

DataType

Description

A `NumericalVectorValue` is a kind of `VectorValue` that is specifically represented as a one-dimensional `Array` of `NumericalValues`. The dimension is allowed to be empty, permitting a `NumericalVectorValue` of rank 0, which is essentially isomorphic to a scalar `NumericalValue`.

General Types

Array
VectorValue

Features

dimension : Positive [0..1] {redefines dimensions}

elements : NumericalValue [0..*] {redefines elements}

Constraints

None.

8.20.2.4 ThreeVectorValue

Element

DataType

Description

A `ThreeVectorValue` is a `NumericalVectorValue` that has dimension 3.

General Types

NumericalVectorValue

Features

dimension : Positive [0..*] {redefines elements}

Constraints

None.

8.20.2.5 VectorValue

Element

DataType

Description

A *VectorValue* is an abstract data type whose values may be operated on using *VectorFunctions*.>/p>

General Types

None.

Features

None.

Constraints

None.

8.21 Base Functions

8.21.1 Base Functions Overview

This package defines a basic set of Functions defined on all kinds of values. Most correspond to similarly named operators in the KerML expression notation.

8.21.2 Elements

```
abstract function '=='(x: Anything[0..1], y: Anything[0..1]): Boolean[1];
function '!='(x: Anything[0..1], y: Anything[0..1]): Boolean[1];

function ToString(x: Anything[0..1]): String;

function '['(
    seq: Anything[0..*] ordered nonunique,
    index: Anything[0..*] ordered nonunique):
    Anything[0..1];
function ','(
    seq1: Anything[0..*] ordered nonunique,
    seq2: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique;

abstract function 'all'(): Object[0..*] {
    abstract feature all 'type': Object;
}

abstract function 'istype'(x: Anything[1]): Boolean[1] {
    abstract feature 'type': Anything;
}
```

```
abstract function '@'(x: Anything[1]): Boolean[1] {
    abstract feature 'type': Anything;
}

abstract function 'hastype'(x: Anything[1]): Boolean {
    abstract feature 'type': Anything[1];
}

abstract function 'as'(
    seq: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique {
    abstract feature 'type': Anything[1];
}
```

8.22 Data Functions

8.22.1 Data Functions Overview

This package defines the abstract base Functions corresponding to all the unary and binary operators in the KerML expression notation that might be defined on various kinds of DataValues.

8.22.2 Elements

```
abstract function '+'(x: DataValue[1], y: DataValue[0..1]): DataValue[1];
abstract function '-'(x: DataValue[1], y: DataValue[0..1]): DataValue[1];
abstract function '*'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function '/'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function '**'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function '^'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function '%'(x: DataValue[1], y: DataValue[1]): DataValue[1];

abstract function '!'(x: DataValue[1]): DataValue[1];
abstract function 'not'(x: DataValue[1]): DataValue[1];
abstract function '~'(x: DataValue[1]): DataValue[1];

abstract function '|'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function '^^'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function 'xor'(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function '&'(x: DataValue[1], y: DataValue[1]): DataValue[1];

abstract function '<'(x: DataValue[1], y: DataValue[1]): Boolean[1];
abstract function '>'(x: DataValue[1], y: DataValue[1]): Boolean[1];
abstract function '<='(x: DataValue[1], y: DataValue[1]): Boolean[1];
abstract function '>='(x: DataValue[1], y: DataValue[1]): Boolean[1];

abstract function max(x: DataValue[1], y: DataValue[1]): DataValue[1];
abstract function min(x: DataValue[1], y: DataValue[1]): DataValue[1];

abstract function '==' specializes BaseFunctions::'=='
    (x: DataValue[0..1], y: DataValue[0..1]): Boolean[1];

abstract function '..'
    (lower: DataValue[1], upper: DataValue[1]): DataValue[0..*] ordered;
```

8.23 Scalar Functions

8.23.1 Scalar Functions Overview

This package defines abstract functions that specialize the DataFunctions for use with ScalarValues.

8.23.2 Elements

```
abstract function '+' specializes DataFunctions::'+'
    (x: ScalarValue[1], y: ScalarValue[0..1]): ScalarValue[1];
abstract function '-' specializes DataFunctions::'-'
    (x: ScalarValue[1], y: ScalarValue[0..1]): ScalarValue[1];
abstract function '*' specializes DataFunctions::'*'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function '/' specializes DataFunctions::'/'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function '**' specializes DataFunctions::'**'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function '^' specializes DataFunctions::'^'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function '%' specializes DataFunctions::'%'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];

abstract function '!' specializes DataFunctions::'!'
    (x: ScalarValue[1]): ScalarValue[1];
abstract function 'not' specializes DataFunctions::'not'
    (x: ScalarValue[1]): ScalarValue[1];
abstract function '~' specializes DataFunctions::'~'
    (x: ScalarValue[1]): ScalarValue[1];

abstract function '|' specializes DataFunctions::'|'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function '^^' specializes DataFunctions::'^^'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function 'xor' specializes DataFunctions::'xor'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function '&' specializes DataFunctions::'&'
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];

abstract function '<' specializes DataFunctions::'<'
    (x: ScalarValue[1], y: ScalarValue[1]): Boolean[1];
abstract function '>' specializes DataFunctions::'>'
    (x: ScalarValue[1], y: ScalarValue[1]): Boolean[1];
abstract function '<=' specializes DataFunctions::'<='
    (x: ScalarValue[1], y: ScalarValue[1]): Boolean[1];
abstract function '>=' specializes DataFunctions::'>='
    (x: ScalarValue[1], y: ScalarValue[1]): Boolean[1];

abstract function max specializes DataFunctions::max
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];
abstract function min specializes DataFunctions::min
    (x: ScalarValue[1], y: ScalarValue[1]): ScalarValue[1];

abstract function '..' specializes DataFunctions::'..'
    (lower: ScalarValue[1], upper: ScalarValue[1]): ScalarValue[0..*];
```

8.24 Boolean Functions

8.24.1 Boolean Functions Overview

This package defines functions on Boolean values, including those corresponding to (non-conditional) logical operators in the KerML expression notation.

8.24.2 Elements

```
function '!' specializes ScalarFunctions::'!'
    (x: Boolean[1]): Boolean[1];
function 'not' specializes ScalarFunctions::'not'
    (x: Boolean[1]): Boolean[1];

function '|' specializes ScalarFunctions::'|'
    (x: Boolean[1], y: Boolean[1]): Boolean[1];
function '^^' specializes ScalarFunctions::'^^'
    (x: Boolean[1], y: Boolean[1]): Boolean[1];
function 'xor' specializes ScalarFunctions::'xor'
    (x: Boolean[1], y: Boolean[1]): Boolean[1];
function '&' specializes ScalarFunctions::'&'
    (x: Boolean[1], y: Boolean[1]): Boolean[1];

function '==' specializes DataFunctions::'=='
    (x: Boolean[0..1], y: Boolean[0..1]): Boolean[1];

function ToString specializes BaseFunctions::ToString
    (x: Boolean[1]): String[1];
function ToBoolean(x: String[1]): Boolean[1];
```

8.25 String Functions

8.25.1 String Functions Overview

This package defines functions on String values, including those corresponding to string concatenation and comparison operators in the KerML expression notation.

8.25.2 Elements

```
function '+' specializes ScalarFunctions::'+'
    (x: String[1], y:String[1]): String[1];

function Length(x: String[1]): Natural[1];
function Substring
    (x: String[1], lower: Integer[1], upper: Integer[1]): String[1];

function '<' specializes ScalarFunctions::'<'
    (x: String[1], y: String[1]): Boolean[1];
function '>' specializes ScalarFunctions::'>'
    (x: String[1], y: String[1]): Boolean[1];
function '<=' specializes ScalarFunctions::'<='
    (x: String[1], y: String[1]): Boolean[1];
function '>=' specializes ScalarFunctions::'>='
    (x: String[1], y: String[1]): Boolean[1];

function '==' specializes DataFunctions::'=='
    (x: String[0..1], y: String[0..1]): Boolean[1];

function ToString specializes BaseFunctions::ToString
    (x: String[1]): String[1] ;
```

8.26 Numerical Functions

8.26.1 Numerical Functions Overview

This package defines abstract Functions on Numerical values for general arithmetic and comparison operations.

8.26.2 Elements

```
abstract function isZero(x: NumericalValue[1]): Boolean;
abstract function isUnit(x : NumericalValue[1]): Boolean;

abstract function abs(x: NumericalValue[1]): NumericalValue[1];

abstract function '+' specializes ScalarFunctions::'+'
    (x: NumericalValue[1], y: NumericalValue[0..1]): NumericalValue[1];
abstract function '-' specializes ScalarFunctions::'-'
    (x: NumericalValue[1], y: NumericalValue[0..1]): NumericalValue[1];
abstract function '*' specializes ScalarFunctions::'*'
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];
abstract function '/' specializes ScalarFunctions::'/'
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];
abstract function '**' specializes ScalarFunctions::'**'
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];
abstract function '^' specializes ScalarFunctions::'^'
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];
abstract function '%' specializes ScalarFunctions::'%'
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];

abstract function '<' specializes ScalarFunctions::'<'
    (x: NumericalValue[1], y: NumericalValue[1]): Boolean[1];
abstract function '>' specializes ScalarFunctions::'>'
    (x: NumericalValue[1], y: NumericalValue[1]): Boolean[1];
abstract function '<=' specializes ScalarFunctions::'<='
    (x: NumericalValue[1], y: NumericalValue[1]): Boolean[1];
abstract function '>=' specializes ScalarFunctions::'>='
    (x: NumericalValue[1], y: NumericalValue[1]): Boolean[1];

abstract function max specializes ScalarFunctions::max
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];
abstract function min specializes ScalarFunctions::min
    (x: NumericalValue[1], y: NumericalValue[1]): NumericalValue[1];

abstract function sum
    (collection: NumericalValue[0..*]): NumericalValue[1];
abstract function product
    (collection: NumericalValue[0..*]): NumericalValue[1];
```

8.27 Complex Functions

8.27.1 Complex Functions Overview

This package defines Functions on Complex values, including concrete specializations of the general arithmetic and comparison operations.

8.27.2 Elements

```
feature i: Complex[1] = Rect(0.0, 1.0);

function rect(re: Real[1], im: Real[1]): Complex[1];
function polar(abs: Real[1], arg: Real[1]): Complex[1];

function isZero specializes NumericalFunctions::isZero (x : Complex);
function isUnit specializes NumericalFunctions::isUnit (x : Complex);

function abs specializes NumericalFunctions::abs
```

```
        (x: Complex[1]): Real[1];
    function arg(x: Complex[1]): Real[1];

    function '+' specializes NumericalFunctions::'+'
        (x: Complex[1], y: Complex[0..1]): Complex[1];
    function '-' specializes NumericalFunctions::'-'
        (x: Complex[1], y: Complex[0..1]): Complex[1];
    function '*' specializes NumericalFunctions::'*'
        (x: Complex[1], y: Complex[1]): Complex[1];
    function '/' specializes NumericalFunctions::'/'
        (x: Complex[1], y: Complex[1]): Complex[1];
    function '**' specializes NumericalFunctions::'**'
        (x: Complex[1], y: Complex[1]): Complex[1];
    function '^' specializes NumericalFunctions::'^'
        (x: Complex[1], y: Complex[1]): Complex[1];

    function '==' specializes DataFunctions::'=='
        (x: Complex[0..1], y: Complex[0..1]): Boolean[1];

    function ToString specializes BaseFunctions::ToString
        (x: Complex[1]): String[1];
    function ToComplex(x: String[1]): Complex[1];

    function sum specializes NumericalFunctions::sum
        (collection: Complex[0..*]): Complex[1];
    function product specializes NumericalFunctions::product
        (collection: Complex[0..*]): Complex[1];
```

8.28 Real Functions

8.28.1 Real Functions Overview

This package defines Functions on Real values, including concrete specializations of the general arithmetic and comparison operations.

8.28.2 Elements

```
    function re :> ComplexFunctions::re(x: Real[1]): Real[1];
    function im :> ComplexFunctions::im(x: Real[1]): Real[1];

    function abs specializes NumericalFunctions::abs
        (x: Real[1]): Real[1];
    function arg specializes ComplexFunctions::arg
        (x: Real[1]): Real[1]

    function '+' specializes ComplexFunctions::'+'
        (x: Real[1], y: Real[0..1]): Real[1];
    function '-' specializes ComplexFunctions::'-'
        (x: Real[1], y: Real[0..1]): Real[1];
    function '*' specializes ComplexFunctions::'*'
        (x: Real[1], y: Real[1]): Real[1];
    function '/' specializes ComplexFunctions::'/'
        (x: Real[1], y: Real[1]): Real[1];
    function '**' specializes ComplexFunctions::'**'
        (x: Real[1], y: Real[1]): Real[1];
    function '^' specializes ComplexFunctions::'^'
        (x: Real[1], y: Real[1]): Real[1];

    function '<' specializes NumericalFunctions::'<'
```

```
        (x: Real[1], y: Real[1]): Boolean[1];
    function '>' specializes NumericalFunctions::'>'
        (x: Real[1], y: Real[1]): Boolean[1];
    function '<=' specializes NumericalFunctions::'<='
        (x: Real[1], y: Real[1]): Boolean[1];
    function '>=' specializes NumericalFunctions::'>='
        (x: Real[1], y: Real[1]): Boolean[1];

    function max specializes NumericalFunctions::max
        (x: Real[1], y: Real[1]): Real[1];
    function min specializes NumericalFunctions::min
        (x: Real[1], y: Real[1]): Real[1];

    function '==' specializes ComplexFunctions::'=='
        (x: Real[0..1], y: Real[0..1]): Boolean[1];

    function sqrt(x: Real[1]): Real[1];

    function floor(x: Real[1]): Integer[1];
    function round(x: Real[1]): Integer[1];

    function ToString specializes ComplexFunctions::ToString
        (x: Real[1]): String[1];
    function ToInteger(x: Real[1]): Integer[1];
    function ToRational(x: Real[1]): Rational[1];
    function ToReal(x: String[1]): Real[1];

    function sum specializes ComplexFunctions::sum
        (collection: Real[0..*]): Real;
    function product specializes ComplexFunctions::product
        (collection: Real[0..*]): Real;
```

8.29 Rational Functions

8.29.1 Rational Functions Overview

This package defines Functions on Rational values, including concrete specializations of the general arithmetic and comparison operations.

8.29.2 Elements

```
    function rat
        (numer: Integer[1], denum: Integer[1]): Rational[1];
    function numer(rat: Rational[1]): Integer[1];
    function denom(rat: Rational[1]): Integer[1];

    function abs specializes RealFunctions::abs
        (x: Rational[1]): Rational[1];

    function '+' specializes RealFunctions::'+'
        (x: Rational[1], y: Rational[0..1]): Rational[1];
    function '-' specializes RealFunctions::'-'
        (x: Rational[1], y: Rational[0..1]): Rational[1];
    function '*' specializes RealFunctions::'*'
        (x: Rational[1], y: Rational[1]): Rational[1];
    function '/' specializes RealFunctions::'/'
        (x: Rational[1], y: Rational[1]): Rational[1];
    function '**' specializes RealFunctions::'**'
        (x: Rational[1], y: Rational[1]): Rational[1];
```

```
function '^' specializes RealFunctions::'^'
    (x: Rational[1], y: Rational[1]): Rational[1];

function '<' specializes RealFunctions::'<'
    (x: Rational[1], y: Rational[1]): Boolean[1];
function '>' specializes RealFunctions::'>'
    (x: Rational[1], y: Rational[1]): Boolean[1];
function '<=' specializes RealFunctions::'<='
    (x: Rational[1], y: Rational[1]): Boolean[1];
function '>=' specializes RealFunctions::'>='
    (x: Rational[1], y: Rational[1]): Boolean[1];

function max specializes RealFunctions::max
    (x: Rational[1], y: Rational[1]): Rational[1];
function min specializes RealFunctions::min
    (x: Rational[1], y: Rational[1]): Rational[1];

function '==' specializes RealFunctions::'=='
    (x: Rational[0..1], y: Rational[0..1]): Boolean[1];

function gcd(x: Rational[1], y: Rational[1]): Integer[1];

function floor(x: Rational[1]): Integer[1];
function round(x: Rational[1]): Integer[1];

function ToString specializes RealFunctions::ToString
    (x: Rational[1]): String[1];
function ToInteger(x: Rational[1]): Integer[1];
function ToRational(x: String[1]): Rational[1];

function sum specializes RealFunctions::sum
    (collection: Rational[0..*]): Rational[1];
function product specializes RealFunctions::product
    (collection: Rational[0..*]): Rational[1];
```

8.30 Integer Functions

8.30.1 Integer Functions Overview

This package defines Functions on Integer values, including concrete specializations of the general arithmetic and comparison operations.

8.30.2 Elements

```
function abs specializes RationalFunctions::abs
    (x: Integer[1]): Natural[1];

function '+' specializes RationalFunctions::'+'
    (x: Integer[1], y: Integer[0..1]): Integer[1];
function '-' specializes RationalFunctions::'-'
    (x: Integer[1], y: Integer[0..1]): Integer[1];
function '*' specializes RationalFunctions::'*'
    (x: Integer[1], y: Integer[1]): Integer[1];
function '/' specializes RationalFunctions::'/'
    (x: Integer[1], y: Integer[1]): Rational;
function '**' specializes RationalFunctions::'**'
    (x: Integer[1], y: Natural): Integer[1];
function '^' specializes RationalFunctions::'^'
    (x: Integer[1], y: Natural): Integer[1];
```

```
function '%' specializes NumericalFunctions::'%'
    (x: Integer[1], y: Integer[1]): Integer[1];

function '<' specializes RationalFunctions::'<'
    (x: Integer[1], y: Integer[1]): Boolean[1];
function '>' specializes RationalFunctions::'>'
    (x: Integer[1], y: Integer[1]): Boolean[1];
function '<=' specializes RationalFunctions::'<='
    (x: Integer[1], y: Integer[1]): Boolean[1];
function '>=' specializes RationalFunctions::'>='
    (x: Integer[1], y: Integer[1]): Boolean[1];

function max specializes RationalFunctions::max
    (x: Integer[1], y: Integer[1]): Integer[1];
function min specializes RationalFunctions::min
    (x: Integer[1], y: Integer[1]): Integer[1];

function '==' specializes RationalFunctions::'=='
    (x: Integer[0..1], y: Integer[0..1]): Boolean[1];

function '..' specializes ScalarFunctions::'..'
    (lower: Integer[1], upper: Integer[1]): Integer[0..*];

function ToString specializes RationalFunctions::ToString
    (x: Integer[1]): String[1];
function ToNatural(x: Integer[1]): Natural[1];
function ToInteger(x: String[1]): Integer[1];

function sum specializes RationalFunctions::sum
    (collection: Integer[0..*]): Integer[1];
function product specializes RationalFunctions::product
    (collection: Integer[0..*]): Integer[1];
```

8.31 Natural Functions

8.31.1 Natural Functions Overview

This package defines Functions on Natural values, including concrete specializations of the general arithmetic and comparison operations.

8.31.2 Elements

```
function '+' specializes IntegerFunctions::'+'
    (x: Natural[1], y: Natural[0..1]): Natural[1];
function '*' specializes IntegerFunctions::'*'
    (x: Natural[1], y: Natural[1]): Natural[1];
function '/' specializes IntegerFunctions::'/'
    (x: Natural[1], y: Natural[1]): Natural[1];
function '%' specializes IntegerFunctions::'%'
    (x: Natural[1], y: Natural[1]): Natural[1];

function '<' specializes IntegerFunctions::'<'
    (x: Natural[1], y: Natural[1]): Boolean[1];
function '>' specializes IntegerFunctions::'>'
    (x: Natural[1], y: Natural[1]): Boolean[1];
function '<=' specializes IntegerFunctions::'<='
    (x: Natural[1], y: Natural[1]): Boolean[1];
function '>=' specializes IntegerFunctions::'>='
    (x: Natural[1], y: Natural[1]): Boolean[1];
```

```
function max specializes IntegerFunctions::max
    (x: Natural[1], y: Natural[1]): Natural[1];
function min specializes IntegerFunctions::min
    (x: Natural[1], y: Natural[1]): Natural[1];

function '==' specializes IntegerFunctions::'=='
    (x: Natural[0..1], y: Natural[0..1]): Boolean[1];

function ToString specializes IntegerFunctions::ToString
    (x: Natural[1]): String[1];
function ToNatural(x: String[1]): Natural[1];
```

8.32 Trig Functions

8.32.1 Trig Functions Overview

This package defines basic trigonometric functions on real numbers.

8.32.2 Elements

```
feature pi : Real;
inv piPrecision {
    RealFunctions::round(pi * 1E20)
        == 314159265358979323846.0 }

function deg(theta_rad : Real);
function rad(theta_deg : Real);

datatype UnitBoundedReal :> Real;

function sin(theta : Real) : UnitBoundedReal;
function cos(theta : Real) : UnitBoundedReal;
function tan(theta : Real) : Real;
function cot(theta : Real) : Real;

function arcsin(x : UnitBoundedReal) : Real;
function arccos(x : UnitBoundedReal) : Real;
function arctan(x : Real) : Real;
```

8.33 Sequence Functions

8.33.1 Sequence Functions Overview

This package defines Functions that operate on general sequences of values. (For Functions that operate on Collection values, see CollectionFunctions.)

8.33.2 Elements

```
function equals (
    x: Anything[0..*] ordered nonunique,
    y: Anything[0..*] ordered nonunique): Boolean[1];

function size(seq: Anything[0..*] nonunique): Natural[1];
function isEmpty(seq: Anything[0..*] nonunique): Boolean[1];
function notEmpty(seq: Anything[0..*] nonunique): Boolean[1];
function includes(
```

```
        seq1: Anything[0..*] nonunique,
        seq2: Anything[0..*]): Boolean[1];
    function includesOnly(
        seq1: Anything[0..*] nonunique,
        seq2: Anything[0..*] nonunique): Boolean[1];
    function excludes(
        seq1: Anything[0..*] nonunique,
        seq2: Anything[0..*]): Boolean[1];

    function union(
        seq1: Anything[0..*] ordered nonunique,
        seq2: Anything[0..*] ordered nonunique):
        Anything[0..*] ordered nonunique;
    function intersection(
        seq1: Anything[0..*] ordered nonunique,
        seq2: Anything[0..*] ordered nonunique):
        Anything[0..*] ordered nonunique;
    function including(
        seq1: Anything[0..*] ordered nonunique,
        seq2: Anything[0..*]):
        Anything[0..*] ordered nonunique;
    function excluding(
        seq1: Anything[0..*] ordered nonunique,
        seq2: Anything[)..*]):
        Anything[0..*] ordered nonunique;

    function subsequence(
        seq: Anything[0..*] ordered nonunique,
        startIndex: Positive[1],
        endIndex: Positive[1]): Anything[0..*];
    function head(eq: Anything[0..*] ordered nonunique):
        Anything[0..1];
    function tail(seq: Anything[0..*] ordered nonunique):
        Anything[0..*] ordered nonunique;
    function last(seq: Anything[0..*] ordered nonunique):
        Anything[0..1];

    function '[' specializes BaseFunctions::'['
        (seq: Anything[0..*] ordered nonunique, index: Positive[1]):
        Anything[0..1];
```

8.34 Collection Functions

8.34.1 Collection Functions Overview

This package defines Functions on Collections (as defined in the Collections package). For Functions on general sequences of values, see the SequenceFunctions package.

8.34.2 Elements

```
    function '==' specializes BaseFunctions::'=='
        (col1: Collection[0..1], col2: Collection[0..1]): Boolean[1];
    function size (col: Collection[1]): Natural[1];
    function isEmpty (col: Collection[1]): Boolean[1];
    function notEmpty (col: Collection[1]): Boolean[1];
    function contains(col: Collection[1], value: Anything[1]): Boolean[1];
    function containsAll(col1: Collection[1], col2: Collection[1]): Boolean[1];

    function head(col: OrderedCollection[1]): Anything[0..1];
```

```
function tail(col: OrderedCollection[1]): Anything[0..*] ordered nonunique;
function last(col: OrderedCollection[1]): Anything[0..1];

function '[' specializes BaseFunctions::'['
    (col: OrderedCollection[1], index: Positive[1]): Anything[0..1];
function 'array[' specializes '['
    (arr: Array[1], indexes: Positive[n] ordered nonunique): Anything[0..1] {
    private feature n : Natural[1] = arr.rank;
}
```

8.35 Vector Functions

8.35.1 Vector Functions Overview

This package defines abstract functions on `VectorValues` corresponding to the algebraic operations provided by a vector space with inner product. It also includes concrete implementations of these functions specifically for `CartesianVectorValues`.

8.35.2 Elements

```
abstract function isZeroVector(v: VectorValue[1]): Boolean[1];

abstract function '+' specializes DataFunctions::'+'
    (v: VectorValue[1], w: VectorValue[0..1]) u: VectorValue[1];
abstract function '-' specializes DataFunctions::'-'
    (v: VectorValue[1], w: VectorValue[0..1]) u: VectorValue[1];

abstract function sum0
    (coll: VectorValue[*] nonunique, zero: VectorValue[1])
    s: VectorValue[1];

function VectorOf
    (components: NumericalValue[1..*] ordered nonunique):
    NumericalVectorValue;

abstract function scalarVectorMult specializes DataFunctions::'*'
    (x: NumericalValue[1], v: NumericalVectorValue[1])
    w: NumericalVectorValue[1];
alias '*' for scalarVectorMult;
abstract function vectorScalarMult specializes DataFunctions::'*'
    (v: NumericalVectorValue[1], x: NumericalValue[1])
    w: NumericalVectorValue[1];
abstract function vectorScalarDiv specializes DataFunctions::'/'
    (v: NumericalVectorValue[1], x: NumericalValue[1])
    w: NumericalVectorValue[1];
abstract function inner specializes DataFunctions::'*'
    (v: NumericalVectorValue[1], w: NumericalVectorValue[1])
    x: NumericalValue[1];

abstract function norm
    (v: NumericalVectorValue[1]) l : NumericalValue[1];
abstract function angle
    (v: NumericalVectorValue[1], w: NumericalVectorValue[1])
    theta: NumericalValue[1];

function CartesianVectorOf
    (components: Real[*]): CartesianVectorValue;
function CartesianThreeVectorOf specializes CartesianVectorOf
```

```
    (components: Real[3]): CartesianThreeVectorValue;

feature cartesianZeroVector: CartesianVectorValue[3] =
    (
        CartesianVectorOf(0.0),
        CartesianVectorOf((0.0, 0.0)),
        CartesianThreeVectorOf((0.0, 0.0, 0.0))
    );
feature cartesian3DZeroVector: CartesianThreeVectorValue[1] =
    cartesianZeroVector[3];

function isCartesianZeroVector specializes isZeroVector
    (v: CartesianVectorValue): Boolean;

function 'cartesian+' specializes '+'
    (v: CartesianVectorValue[1], w: CartesianVectorValue[0..1]):
    u: CartesianVectorValue[1];
function 'cartesian-' specializes '-'
    (v: CartesianVectorValue[1], w: CartesianVectorValue[0..1]):
    u: CartesianVectorValue[1];
function cartesianScalarVectorMult specializes scalarVectorMult
    (x: Real[1], v: CartesianVectorValue[1])
    w: CartesianVectorValue[1];
function cartesianVectorScalarMult specializes vectorScalarMult
    (v: CartesianVectorValue[1], x: Real[1])
    w: CartesianVectorValue[1];
    function cartesianInner specializes inner
    (v: CartesianVectorValue[1], w : CartesianVectorValue[1])
    x: Real[1];

function cartesianNorm specializes norm
    (v: CartesianVectorValue[1]) l: NumericalValue[1];
function cartesianAngle specializes angle
    (v: CartesianVectorValue[1], w: CartesianVectorValue[1])
    theta: Real[1];

function sum(coll: CartesianThreeVectorValue): CartesianThreeVectorValue;
```

8.36 Control Functions

8.36.1 Control Functions Overview

This package defines Functions that correspond to operators in the KerML expression notation for which one or more operands are Expressions whose evaluation is determined by another operand.

8.36.2 Elements

```
abstract function '.' {
    in feature source : Anything[0..*] nonunique {
        abstract feature target : Anything[0..*] nonunique;
    }
    return : Anything[0..*] nonunique;
}

abstract function '?'(test: Boolean[1]): Anything[0..*] ordered nonunique {
    abstract composite expr thenValue[0..1] (): Anything[0..*] ordered nonunique;
    abstract composite expr elseValue[0..1] (): Anything[0..*] ordered nonunique;
}
```

```
abstract function '??'(
    firstValue: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique {
    abstract composite expr secondValue[0..1] (): Anything[0..*] ordered nonunique;
}

abstract function '&&'(firstValue: Boolean[1]): Boolean[1] {
    abstract composite expr secondValue[0..1] (): Boolean[1];
}

function 'and'(firstValue: Boolean[1]): Boolean[1] {
    abstract composite expr secondValue[0..1] (): Boolean[1];
}

abstract function '||'(firstValue: Boolean[1]): Boolean[1] {
    abstract composite expr secondValue[0..1] (): Boolean[1];
}

function 'or'(firstValue: Boolean[1]): Boolean[1] {
    abstract composite expr secondValue[0..1] (): Boolean[1];
}

abstract function '=>'(firstValue: Boolean[1]): Boolean[1] {
    abstract composite expr secondValue[0..1] (): Boolean[1];
}

function 'implies'(firstValue: Boolean[1]): Boolean[1] {
    abstract composite expr secondValue[0..1] (): Boolean[1];
}

abstract function collect(
    collection: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique {
    abstract composite expr mapper[0..*] (argument: Anything[1]):
        Anything[0..*] ordered nonunique;
}

abstract function select(
    collection: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique {
    abstract composite expr selector[0..*] (argument: Anything[1]):
        Boolean[1];
}

function selectOne(
    collection: Anything[0..*] ordered nonunique):
    Anything[0..1] {
    abstract composite expr selector1[0..*] (argument: Anything[1]):
        Boolean[1];
}

abstract function reject(
    collection: Anything[0..*] ordered nonunique):
    Anything[0..*] ordered nonunique {
    abstract composite expr rejector[0..*] (argument: Anything[1]):
        Boolean[1];
}

abstract function reduce(
    collection: Anything[0..*] ordered nonunique):
```

```
    Anything[0..*] ordered nonunique {
    abstract composite expr reducer[0..*] (
        firstArg: Anything[1],
        secondArg: Anything[1]): Anything[1];
}

abstract function forAll(
    collection: Anything[0..*] ordered nonunique):
    Boolean[1] {
    abstract composite expr test[0..*] (argument: Anything[1]):
        Boolean[1];
}

abstract function exists(
    collection: Anything[0..*] ordered nonunique):
    Boolean[1] {
    abstract composite expr test[0..*] (argument: Anything[1]):
        Boolean[1];
}

function allTrue(collection: Boolean[0..*]): Boolean[1];
function anyTrue(collection: Boolean[0..*]): Boolean[1];

function minimize(collection: ScalarValue[1..*]): ScalarValue[1] {
    abstract composite expr fn[0..*] (argument: ScalarValue[1]): ScalarValue[1];
}

function maximize(collection: ScalarValue[1..*]): ScalarValue {
    abstract composite expr fn[0..*] (argument: ScalarValue[1]): ScalarValue[1];
}
```

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.

Release Note. Model interchange will be addressed more fully in the final 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

Release Note. The conformance test suite will be provided in the final submission.