An OMG® Systems Modeling Publication

Kernel Modeling Language™ (KerML™)

Version 1.0 Beta 1

OMG Document Number: ptc/2023-06-01

Date: June 2023

Standard document URL: https://www.omg.org/spec/KerML/1.0/

Machine Readable File(s): https://www.omg.org/spec/KerML/20230201/

Normative:

https://www.omg.org/spec/KerML/20230201/KerML.xmi
https://www.omg.org/spec/KerML/20230201/Semantic-Library.kpar
https://www.omg.org/spec/KerML/20230201/Data-Type-Library.kpar
https://www.omg.org/spec/KerML/20230201/Function-Library.kpar
https://www.omg.org/spec/KerML/20230201/KerML-Model-Interchange.json
https://www.omg.org/spec/KerML/20230201/KerML.json

OMG'S ISSUE REPORTING PROCEDURE

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Documents, Report a Bug/Issue.

Table of Contents

List of Tables

List of Figures

0 Preface

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. All OMG Specifications are available from the OMG website at: https://www.omg.org/spec

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

OMG Headquarters
9C Medway Road, PMB 274
Milford, MA 01757
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320

Email: pubs@omg.org

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

Issues

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page https://www.omg.org, under Specifications, Report an Issue.

1 Scope

The Kernel Modeling Language (KerML) is an application-independent modeling language with a well-grounded formal semantics for modeling existing or planned systems. The language includes general syntactic constructs for structuring models, such as relationships, annotations and namespaces; core semantic constructs that have semantics based on classification; and additional constructs for commonly needed modeling capabilities, such as associations and behaviors.

System models are expressed in KerML using a textual concrete syntax. This can be parsed to an abstract syntax representation, which is then given a semantic interpretation for the system being modeled. The semantics for the KerML core constructs is grounded in formal mathematical logic, providing a consistent basis for mathematical reasoning about KerML models. However, beyond this, the semantics of KerML constructs are specified by the relationship of user model elements to the KerML Semantic Library.

The Semantic Library models, also expressed in KerML, provide an ontological model of the meaning of KerML models. Indeed, all KerML models can be semantically expressed using solely core modeling constructs referencing the appropriate semantic concepts defined in the Semantic Library. KerML semantic constructs beyond the core are essentially just syntactic conveniences for reusing specific library concepts: structures for modeling *objects*, behaviors for modeling *performances*, associations for modeling *links*, etc.

Indeed, the full KerML language can be considered to be simply a syntactic extension of the core, which is semantically extended using library models. By intent, this approach can also be used to build on KerML to create more specific modeling languages. Application specific modeling languages can be built on KerML by extending the KerML abstract syntax, specializing its semantics, with concrete syntaxes similar to or entirely different from KerML's.

To support this, the KerML Semantic Library also includes additional library models beyond those directly providing semantics for KerML syntactic constructs, capturing 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.

In this way, KerML can provide the kernel for a family of syntactically diverse but semantically integrated modeling languages.

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 8. 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 8) into an abstract syntax form, or may be represented only in an abstract syntax form (see also 8.2 and 8.3).

A *KerML modeling tool* is a software application 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 8.)

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 8.)

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 8 and Clause 9. 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) as specified in Clause 10.

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.

3 Normative References

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

[ADLER] *ZLIB Compressed Data Format Specification*, Version 3.3
https://datatracker.ietf.org/doc/html/rfc1950

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

[BLAKE] *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*
https://www.rfc-editor.org/rfc/rfc7693
BLAKE3
https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf

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

[ISO8601] *ISO 8601-1:2019 (First edition) Date and time – Representations for information interchange — Part 1: Basic rules*
https://www.iso.org/standard/70907.html

[ISO10646] *ISO/IEC 10646:2010 (Second edition) Information technology – Universal Coded Character Set (UCS)*

[JSON] *ISO/IEC 21778:2017 Information technology – The JSON data interchange syntax*
https://www.iso.org/standard/71616.html
(see also *IECMA-404 The JSON data interchange syntax*
https://www.ecma-international.org/publications-and-standards/standards/ecma-404/)

[MD] The MD2 Message-Digest Algorithm
https://datatracker.ietf.org/doc/html/rfc1319
The MD4 Message-Digest Algorithm
https://www.rfc-editor.org/rfc/rfc1320
The MD5 Message-Digest Algorithm
https://www.rfc-editor.org/rfc/rfc1321

[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

[SHS] *FIPS Pub 180-4 Secure Hash Standard*
https://csrc.nist.gov/publications/detail/fips/180/4/final

[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] *ITU-T X.667 (10/2012) Information technology – Procedures for the operation of object identifier registration authorities: Generation of universally unique identifiers and their use in object identifiers*
https://www.itu.int/rec/T-REC-X.667-201210-I
(see also *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

[ZIP] *.ZIP File Format Specification*
https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

4 Terms and Definitions

Various terms and definitions are specified throughout the body of this specification.

5 Symbols

A concrete syntax for KerML is specified in subclause 8.2 of this specification.

6 Introduction

6.1 Language Architecture

Developing systems generally involves creating a number of different specifications. For instance, a requirements specification gives the intended effects of a system, while a design specification determines how the system will bring about those effects. Many designs might be developed and evaluated against the same requirements. A test specification then describes test procedures that check whether requirements are met by real or virtual systems built and operated according to some design.

A *model* is a representation in some *modeling language* of all or part of any of the above kinds of system specification. The *semantics* of such models defines what it means for real or virtual things in a modeled system to conform to the specification given by the model. KerML is a foundational modeling language for expressing various kinds of system models with consistent semantics.

Syntactically, KerML is divided into three layers, with each layer building increasingly specific constructs on the previous layer. These layers are, from general to specific:

1. The *Root Layer* includes the most general syntactic constructs for structuring models, such as elements, relationships, annotations, and packaging.
2. The *Core Layer* includes the most general constructs that have semantics based on *classification*.
3. The *Kernel Layer* provides commonly needed modeling capabilities, such as associations and behavior.

The Core Layer grounds KerML semantics by interpreting it using mathematical logic. However, additional semantics are then specified through the relationship of Kernel abstract syntax constructs to model elements in the *Kernel Semantic Library,* which is written in KerML itself. Models expressed in KerML thus essentially reuse elements of the Semantic Library to give them semantics. The Semantic Library models give the basic conditions for the conformance of modeled things to the model, which are then augmented in the user model as appropriate.

Having a consistent specification of semantics helps people interpret models in the same way. In particular, because the Semantic Library models are expressed in the same language as user models, engineers and tool builders can inspect the library models to formally understand what real or virtual effects are actually being specified by their models for systems being modeled. More uniform model interpretation improves communication between everyone involved in modeling, including modelers and tool builders.

6.2 Document Organization

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

- Clause 7 describes KerML from a user point of view, covering all the modeling constructs in the language. It is an informative reference for the normative language specification given in the following three subclauses.
- Clause 8 specifies the normative metamodel for the KerML language. This includes the complete grammar for the concrete syntax, which is a textual notation (see 8.2), the abstract syntax, which is a MOF model (see 8.3), and formal semantics (see 8.4).
- Clause 9 specifies the normative Kernel Model Libraries, each of which is a set of *library models* available to be used in all KerML user models. They include the Semantic Library, which is a set of KerML models used to provide Kernel-layer semantics to user models (see 9.2), the Data Type Library of standard data types (see 9.3) and the Function Library of functions on those data types (see 9.4).
- Clause 10 specifies the format for standard file-based interchange of KerML models between tools.

In addition, Annex A provides basic (non-normative) guidance on incrementally instantiating models for execution, in a way that conforms to the formal semantics as (normatively) specified in the metamodel (see 8.4), as supported by the Semantic Model Library (see 9.2).

6.3 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
- Ivan Gomes, Twingineer
- Hans Peter de Koning, DEKonsult
- Vince Molnár, Budpest University of Technology and Economics

Other contributors include:

- Manfred Koethe, 88solutions
- Karen Ryan, Siemens

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 200 people in over 80 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, Twingineer
- Doris Lam, Jet Propulsion Laboratory
- Robert Karban, Jet Propulsion Laboratory
- Christopher Klotz, No Magic/Dassault Systèmes
- John Watson, Lightstreet Consulting

7 Language Description

(Informative)

7.1 Language Description Overview

This clause provides an informative description of KerML. Clause 8 gives the full definition of the KerML metamodel, which is the normative specification for implementing the language. In contrast, the description in this clause focuses on how the various constructs of the language are used, along with the Kernel Model Library (see Clause 9), to construct models. While non-normative, it is intended to be precise and consistent with the normative specification of the language.

The following subclauses present the language features in each of the Root, Core and Kernel Layers of KerML (as described in 6.1). Each layer is then further subdivided, following a parallel structure to the packaging of the metamodel (see 8.1). Each subclause within a layer includes references to the corresponding concrete syntax, abstract syntax and semantics subclauses from the normative metamodel specification. In this way, the clause can be used as a general reference for KerML as well as a guide for better understanding of the formal specification of the metamodel.

This clause contains many examples of the KerML textual notation. In order to distinguish this text from normal body text, the following stylistic conventions are used in this clause.

1. Textual notation appears in "`code`" font. This includes references to individual element names from both example models (such as `Vehicle` and `wheels`) and the Kernel Model Library (such as `Performance` and `performances`), as well as more extensive model snippets.
2. Keywords appear in **`boldface`**, both when referenced in-line in body text ("Features are declared using the **`feature`** keyword.") and when used within complete notation examples.
3. Longer samples of textual notation are written in separate paragraphs, indented relative to body paragraphs.

7.2 Root

7.2.1 Root Overview

The Root layer provides the most general syntactic capabilities of the language: elements and relationships between them, annotations of elements, and membership of elements in namespaces. These capabilities are the syntactic foundation for structuring models in KerML, but they do not actually represent anything about a modeled system, and so have no semantic specification. The Core and Kernel layers build on the foundation provided by Root to provide constructs with modeling semantics (see 7.3 and 7.4).

7.2.2 Elements and Relationships

7.2.2.1 Elements and Relationships Overview

Metamodel references:

- *Concrete syntax, 8.2.3.1*
- *Abstract syntax, 8.3.2.1*
- *Semantics, none*

Elements are the constituents of a model. Some elements represent *relationships* between other elements, known as the *related elements* 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 related element in a relationship and for there to be relationships between relationships.

One of the related elements of a relationship may be the *owning* related element of the relationship. If the owning related element of a relationship is deleted from a model, then the relationship is also deleted. Some of the related elements of a relationship (distinct from the owning related element, if any) may be *owned* related elements. If a relationship has owned related elements, then, if the relationship is deleted from a model, all its owned related elements are also deleted.

The *owned relationships* of an element are all those relationships for which the element is the owning related element. The *owned elements* of an element are all those elements that are owned related elements of the owned relationships of the element (notice the extra level of indirection through the owned relationships). The *owning relationship* of an element (if any) is the relationship for which the element is an owned related element (of which the element can have at most one). The *owner* of an element (if any) is the owning related element of the owning relationship of the element (again, notice the extra level of indirection through the owning relationship).

The deletion rules for relationships imply that, if an element is deleted from a model, then all its owned relationships are also deleted and, therefore, all its owned elements. 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.

7.2.2.2 Elements

Every element has a unique identifier known as its *element ID*. The properties of an element can change over its lifetime, but its element ID does not change after the element is created. An element may also have additional identifiers, its *alias IDs,* which may be assigned for tool-specific purposes.

The KerML textual notation, however, does not have any provision for specifying element or alias IDs, since these are expected to be managed by the underlying modeling tooling. Instead, an element may also have a *name* and/or a *short name,* by which it can be referenced in the notation. While the language makes no formal distinction between names and short names, 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 short name, if given, should be an abbreviated name useful for referring to the element. (For further discussion of naming, see also 7.2.5).

In most cases, an element is *declared* using a keyword indicating the *kind* of element it is (e.g., **`classifier`** or **`feature`**). The declaration of an element may also specify a short name and/or name for it, in that order. The short name is distinguished by being surrounded by the delimiting characters < and >.

```
classifier <c123> AClassifier;
feature aFeature;
```

Note that it is not required to specify either a short name 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 notation.

Names and short names have the same lexical structure, which has two variants.

1. A *basic name* is one that can be lexically distinguished in itself from other parts of the notation. The initial character of a basic name must be a lowercase letter, an uppercase letter or an underscore. The remaining characters of a basic name can be any character allowed as an initial character or any digit. However, a reserved keyword may not be used as a name, even though it has the form of a basic name (see 8.2.2.6 for the list of reserved words).

   ```
   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 name consists of the characters *within* the single quotes – the single quotes are *not* included as part of the represented name. 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 within the name by using an escape sequence.

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

An *escape sequence* is a sequence of two text characters starting with a 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). Table 4 in subclause 8.2.2.3 shows the meaning of the allowed escape sequences.

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.

```
namespace P {
    // This is the body of the namespace, declaring its owned members.
    classifier A;
    classifier B {
        // This is the body of the classifier, declaring its owned features.
        feature x;
        feature y;
    }
}
```

7.2.2.3 Relationships

The related elements 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 must have at least two related elements. A relationship with exactly two related elements is known as a *binary relationship*. A *directed binary relationship* is a binary relationship in which one related element is the source and one is the target. Most specialized kinds of relationship in KerML are directed binary relationships (the principal exceptions being dependencies, associations and connectors, see 7.2.3, 7.4.5, and 7.4.6).

Various kinds of relationships are declared with special notations showing their related elements. A relationship may also have a body that specifies owned related elements of the relationship, which may include any kind of element other than an annotating element (see 7.2.4). If an annotating element (i.e., a comment, textual representation or metadata feature) is included in the body of a relationship, then, rather than being directly an owned related element of the containing relationship, the annotating element is an owned related element of an annotation relationship owned by the containing relationship (see 7.2.3.2 for an example).

7.2.3 Dependencies

7.2.3.1 Dependencies Overview

Metamodel references:

- *Concrete syntax, 8.2.3.2*
- *Abstract syntax, 8.3.2.2*
- *Semantics, none*

A *dependency* is a kind of relationship between any number of client (source) and supplier (target) elements. It implies that a change to a supplier element may result in a change to a client element. Dependencies can be useful for representing relationships between elements in an abstract way. For example, a dependency can be used to represent that an upper layer of an architecture stack may depend on a lower layer of the stack.

7.2.3.2 Dependency Declaration

A dependency is declared using the keyword **dependency**, optionally followed by a short name and/or name (see 7.2.2). The client elements of the dependency are then given as a comma-separated list of qualified names following the keyword **from**, followed by a similar list of the supplier elements after the keyword **to**. If no short name or name is given for the dependency, then the keyword **from** may be omitted.

```
dependency Use
    from 'Application Layer' to 'Service Layer';

// 'Service Layer' is the client of this dependency, not its name.
dependency 'Service Layer'
    to 'Data Layer', 'External Interface Layer';
```

A dependency declaration may also optionally have a relationship body (see 7.2.2.3) containing any additional owned related elements (which act as suppliers) and annotating elements owned by the dependency via annotation relationships (see 7.2.4).

```
dependency 'Service Layer'
    to 'Data Layer', 'External Interface Layer' {
    /* 'Service Layer' is the client of this dependency,
     * not its name. */
}
```

7.2.4 Annotations

7.2.4.1 Annotations Overview

Metamodel references:

- *Concrete syntax, 8.2.3.3*
- *Abstract syntax, 8.3.2.3*
- *Semantics, none*

An *annotation* is a relationship between an *annotated element* and an *annotating element* that provides additional information about the element being annotated. Any kind of element may be annotated, but only certain kinds of elements may be annotating elements. Specific kinds of annotating elements include comments and textual representations (see 7.2.4.2 and 7.2.4.3). A further kind of annotating element for user-defined metadata is defined in the Kernel layer (see 7.4.13).

Each annotation relationship is between a single annotating element and a single annotated element, but an annotating element may have multiple annotation relationships with different annotated elements, and any element may have multiple annotations. The annotated element of an annotation can optionally be the owning related element of the annotation, in which case the annotation is an owned annotation of the owning annotated element. If an annotating element is an owned member of a namespace (see 7.2.5) and is not involved in any annotation relationships, then its owning namespace is considered to be its annotated element without the need for an explicit annotation relationship.

7.2.4.2 Comments and Documentation

A *comment* is an annotating element with a textual `body` that in some way describes its annotated element. *Documentation* is a kind of comment that has the special status of documenting the annotated element, known in this case as the *documented element*. A documentation comment is always an owned element of its documented element.

The full declaration of a comment begins with the keyword **`comment`**, optionally followed by a short name and/or name (see 7.2.2.2). One or more annotated elements 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 also 8.2.2.2).

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

If the comment is an owned member of a namespace (see 7.2.5), then the explicit identification of annotated elements can be omitted, in which case the annotated element is implicitly the containing namespace. Further, in this case, if no short name 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. */
}
```

A documentation comment is notated similarly to a regular comment, but using the keyword **`doc`** rather than **`comment`**. The documented element of a documentation comment is always the owning element of the documentation.

```
dependency X from A to B {
    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 does not include the initial `/*` and final `*/` characters. Further, the written text is processed to allow formatting using * characters to delimit consistent initial indentation of a comment lines. For example, the comment notation in:

```
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 tool may (but is not required to) display such text as rendered according to the markup. (See 8.2.3.3.2 for the complete rules for processing comment text.)

7.2.4.3 Textual Representations

A textual representation is an annotating element whose textual `body` represents its annotated element (known in this case as the *represented element*) in a given language. A textual representation is notated similarly to a documentation comment (see 7.2.4.2), but with the keyword **`rep`** used instead of **`comment`**. As for documentation, a textual representation is always owned by its represented element. In particular, if the textual representation is an owned member of a namespace (see 7.2.5), the represented element is the containing Namespace. A textual representation declaration must also specify the language used for the textual body as a literal string (see 8.2.2.5) following the keyword **`language`**. If the textual representation has no short name 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 textual representation is processed as for regular comment text (see 7.2.4.2), and it is the result after such processing that is the textual representation `body` expected to conform to the named language.

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

The language name in a textual representation is case insensitive. The name can be of a natural language, but will often be for a machine-parsable language. In particular, there are recognized standard language names.

If the language is `"kerml"`, then the body of the textual representation must be a legal representation of the represented element in the KerML textual notation. A tool can use such a textual representation to record the original KerML notation text from which an element is parsed. Other standard language names that can be used in a textual representation include `"ocl"` and `"alf"`, in which case the body of the textual representation must be written in the Object Constraint Language [OCL] or the Action Language for fUML [Alf], respectively.

However, for any other language than `"kerml"`, the KerML specification does not define how the body text is to be semantically interpreted as part of the model being represented. An element with no other definition than a textual representation in a language other than KerML is essentially a semantically "opaque" element specified in the other language. Nevertheless, a conforming KerML tool may (but is not required to) interpret such an element consistently with the specification of the named language.

7.2.5 Namespaces

7.2.5.1 Namespaces Overview

Metamodel references:

- *Concrete syntax, 8.2.3.4*
- *Abstract syntax, 8.3.2.4*
- *Semantics, none*

A *namespace* is an element that contains other elements via *membership* relationships with those elements. The namespace is the source element and owner of the membership. The target of a membership can be any kind of element, known as the *member element* of the membership. If the membership is an *owning* membership, then the member element is known as an *owned* member element, which is the only owned related element of the membership.

A namespace may also *import* memberships from other namespaces. Further, a type, which is kind of namespace, may *inherit* memberships from other types that it specializes (see 7.3.2).

The *members* of a namespace are the member elements of all the memberships of the namespace (whether owned, imported or inherited). The *owned members* of a namespace are the owned member elements of all the owned memberships of the namespace that are owning memberships.

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 namespace (see 7.2.5.3), 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 general, 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.

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 (see 8.2.3.5).

Since namespaces and their members may have aliases (see 7.2.5.2), 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.

7.2.5.2 Namespace Declaration

A namespace that is not a root namespace (see 7.2.5.3), 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 short name and/or name (see 7.2.2.2). 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.
}
```

Declaring an element within the body of a namespace denotes that the element is an owned member of the namespace—that is, that there is an owning membership relationship between the namespace and the member element.

The *visibility* of the membership can be specified by placing one of the keywords **public**, **protected** or **private** before the public element declaration. If the membership is **public** (the default), then it is visible outside of the namespace. If it is **private**, then it is not visible. 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 N3 {
    public class C;
    private datatype D;
    feature f : C; // public by default
}
```

An *alias* for an element is a non-owning membership of the element in a namespace, which may or may not be the same namespace that owns the element. An alias name or short name is determined only relative to its membership in the namespace, and can therefore be different than the name or short name defined on the element itself. Note that the same element may be related to a namespace by multiple alias memberships, allowing the element to have multiple, different names relative to that namespace.

An alias is declared using the keyword **alias** followed by the alias short name and/or name, with a qualified name identifying the element given after the keyword **for**. The alias declaration may optionally include a body as described for relationships in 7.2.2.3. The visibility of the alias membership can be specified as for an owned member.

```
namespace N4 {
    class A;
    class B;
    alias <C> CCC for B {
       doc /* Documentation of the alias. */
    }
    private alias D for B;
}
```

A comment (see 7.2.4.2), including documentation, declared within a namespace body also becomes an owned member of the namespace. If no annotated elements are specified for the comment (with an **about** clause), then, by default, the comment is considered to be about the containing namespace.

```
namespace N5 {
    class A;
    comment Comment1 about A
        /* This is a comment about class A. */

    comment Comment2
        /* This is a comment about namespace N5. */

    /* This is also a comment about namespace N5. */
```

```
        doc N9_Doc
            /* This is documentation about namespace N5. */
    }
```

With the ability to specify names, short names and aliases for elements, any element can potentially have several names relative to a namespace. However, the set of names provided for any one member of a namespace must be disjoint from the set of names provided for any other member of the namespace. That is, a namespace effectively provides a "space" of names, each one of which uniquely identifies a single member element of the namespace (though there may be multiple names that identify the same element). This is known as the *distinguishability* of namespace memberships.

7.2.5.3 Root Namespaces

A *root namespace* is a namespace that has no owner. The owned members of a root namespace are known as *top-level elements*. Any element that is not a root namespace has an owner and, therefore, must be in the ownership tree of a top-level element of some root namespace.

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. */
classifier A {
    feature c : C;
}
class C;
datatype D;
feature f: C;
package P;
```

Since the notation does not provide a means for naming a root namespace, the name of a top-level element is *not* qualified by the name of its containing root namespace. The name resolution rules consider all top-level elements to be directly and globally visible without qualification (see 8.2.3.5). 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.

7.2.5.4 Imports

A namespace may *import* visible memberships from other namespaces. The complete set of memberships of a namespace include all its owned memberships and all its imported memberships, and the member elements of imported memberships are included in the set of members of the namespace. Various kinds of namespaces may also define additional memberships to be included in the set of memberships of that kind of namespace (for instance, the memberships of a type also include its *inherited* members – see 7.3.2) and which of those are visible (e.g., public inherited memberships).

If the member name or member short name of any imported membership conflicts with the name of any owned member, or with the name of any visible membership from any other imported namespace, then the conflicting membership is *hidden* and is not included in the set of imported memberships of the importing namespace. As a result of this rule and the distinguishability rule for owned members (see 7.2.5.2), the names of all owned and imported members will always be distinct from each other. Any specialized kind of namespace that adds further kinds of memberships (e.g., inherited memberships of types) always maintains the property that the names of all memberships of a namespace are distinct from each other.

The namespace that is the source of an import relationship, known as the *importing* namespace, also owns it. There are two types of import relationships. A *membership import* is a relationship between the importing namespace and a single membership, which becomes an imported membership of the importing namespace. A *namespace import* is a relationship between the importing namespace and an *imported namespace,* in which all visible memberships of the imported namespace become imported memberships of the importing namespace.

A membership import is denoted using the keyword **`import`** followed by a qualified name, which identifies the imported membership (be member name or member short name). The member element of the imported membership becomes an *imported member* of the importing namespace. Note that the imported membership may be for an alias of the imported member (see 7.2.5.2), in which case the element will be known by that name in the importing namespace.

```
namespace N6 {
    import N4::A;
    import N4::C; // Imported with name "C".
    namespace M {
        import C; // "C" is re-imported from N4 into M.
    }
}
```

A namespace import is also denoted using the keyword **`import`** followed by a qualified name, but with the qualified name suffixed by `"::*"`. In this case, the qualified name identifies the imported namespace. All visible memberships of the imported namespace then become imported memberships of the importing namespace.

```
namespace N7 {
    // Memberships A, B and C are all imported from N4.
    import N4::*;
}
```

If the declaration of either a membership or namespace import is further suffixed by `"::**"`, then the import is *recursive.* Such an import is equivalent to importing memberships as described above for either an imported membership or namespace, followed by further recursively importing from each imported member that is itself a namespace.

```
namespace N8 {
    class A;
    class B;
    namespace M {
        class C;
    }
}
namespace N9 {
    import N8::**;
    // The above recursive import is equivalent to all
    // of the following taken together:
    //     import N8;
    //     import N8::*;
    //     import N8::M::*;
}
namespace N10 {
    import N8::*::**;
    // The above recursive import is equivalent to all
    // of the following taken together:
    //     import N8::*;
    //     import N8::M::*;
    // (Note that N8 itself is not imported.)
}
```

The *visibility* of an import can be specified by placing the keyword **`public`** or **`private`** before the import declaration. If the import is **`public`** (the default), then all the imported memberships become public for the importing namespace. If import is **private**, then the imported memberships become private relative to the importing namespace. An import declaration may optionally have a body, as described for relationships in 7.2.2.3.

```
namespace N11  {
    public import N4::A {
        /* The imported membership is visible outside N11. */
    }

    private import N5::* {
        doc /* None of the imported memberships are visible
             * outside of N11. */
    }
}
```

An import may also be declared with one or more *filter conditions.* Given as model-level evaluable Boolean expressions (see 7.4.9), listed after the imported membership or namespace specification, each surrounded by square brackets [...]. Such a filtered import is equivalent to importing an implicit package that then both imports the given imported membership or namespace and has all the given filter conditions. The effect is such that, for a filtered import, memberships are imported if and only if they satisfy all the given filter conditions. (While filtered imports may be used in any namespace, packages and filter conditions are actually Kernel-layer concepts, because expressions are only defined in that layer. See 7.4.14.)

```
namespace N12 {
    import Annotations::*;

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

7.3 Core

7.3.1 Core Overview

The Core layer builds on the Root layer to add the minimum constructs for modeling systems as designed, built and operated. *Semantics* is about how models are interpreted as giving conditions on how things should be (i.e., as a *specification* of a modeled system) or as a reflection of how things are (i.e, as a *description* of a modeled system). KerML semantics are based on *classification:* a model has elements that classify things in the modeled system.

A *type* is the most general kind of model element that classifies things (see 8.2.4.1.1). *Classifiers* are types that classify things, such as cars, people and processes being carried out, as well as how they are related by features (see 7.3.3). *Features* are also types, classifying relations between things (see 8.2.4.3.1). In addition to simple relations between two things, KerML allows features to classify longer *chains* of relations. 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 relations, first the ownership of the car, then the residence of the owner.

KerML also supports taxonomies of classifications using *specialization* relationships between types. All the things classified by a specialized type are also classified by the general types it is related to via specialization relationships. This means that all the things classified by a specialized type have all the features of its general types, referred to as *inheriting* features from general to specific types. KerML includes several special kinds of specialization, including *subclassification* between classifiers, *subsetting* and *redefinition* between features, and *feature typing* between a feature and another type.

7.3.2 Types

7.3.2.1 Types Overview

Metamodel references:

- *Concrete syntax, 8.2.4.1*
- *Abstract syntax, 8.3.3.1*
- *Semantics, 8.4.3.2*

Types classify things in a modeled system. The set of things classified by a type is the *extent* of the type, each member of which is an *instance* of the type. Everything being modeled is an instance of the type `Anything` from the `Base` library model (see 9.2.2).

A type gives conditions for what things must be in or not in its extent (*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. These conditions apply to all procedures that determine the extents of types, including logical solving, inference, and execution.

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. (Note that necessary conditions are usually stated as what must be true of all instances in the extent, even though they really only determine what is not.) Alternatively, `Car` could require all four wheeled things to be in its extent (sufficient condition).

Types are namespaces, enabling them to have members via membership relationships to other elements identified as their members (see 7.2.5). These include *inherited* memberships, which are certain memberships from the general types of their *owned specializations* (see 7.3.2.3). The member names of all inherited memberships must be distinct from each other and from the member names of all owned memberships. A membership that would otherwise be imported is also hidden by an inherited memberships with the same member name, similarly to how it would be hidden by a conflicting owned membership (see 7.2.5).

Note. Name conflicts due to inherited memberships can be resolved by redefining them to give non-conflicting member names (see 7.3.4).

7.3.2.2 Type Declaration

A type is declared using the keyword **`type`**, optionally followed by a short name and/or name. In addition, a type declaration defines either one or more *owned specializations* for the type (see 7.3.2.3) or a *conjugator* for the type (see 7.3.2.4). This may optionally be followed by the definition of one or more *owned disjoinings* (see 7.3.2.5).

```
type A specializes Base::Anything disjoint from B;
type C conjugates A;
```

A type is specified as *abstract* by placing the keyword **`abstract`** before the keyword **`type`**. A type that is not abstract is called a *concrete* type. Declaring a type to be abstract means that all instances of the type must also be instances of at least one concrete type that directly or indirectly specializes the abstract type.

```
abstract type A specializes Base::Anything;
type A1 specializes A;
type A2 specializes A;
```

The multiplicity constrains the number of instances in the extent of a type (the *cardinality* of the extent). A multiplicity is a feature whose values are natural numbers (extended with infinity, see 9.3.2.1) that are the only ones allowed for the cardinality of its featuring type (each multiplicity is the feature of exactly one Type). A type can have at most one feature that is its multiplicity. Cardinality for classifiers is the number of things it classifies. For features that are not end features (see below), cardinality is the number of values of the feature for a specific instance of its featuring types.

Note. The semantics of multiplicity is different for features that are identified as *end features.* End Features are used primarily in the definition of associations and connectors, and the semantics of end features is discussed in conjunction with them (see 7.4.5 and 7.4.6, respectively).

The multiplicity of a type can be specified as a *range* after any identification of the Type, between square brackets `[...]`. (See 7.4.12 for a complete description of multiplicity ranges, including declaring named multiplicity features.)

```
// 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 7.2.5.2). However, for types, *protected* members, indicated using the keyword **`protected`** instead of **`public`** or **`private`**, have special visibility rules for inheritance (see 7.3.2.3). A feature declared as an owned member of a type is automatically considered to be an *owned feature* of the type, related by a *feature membership*, unless its declaration is preceded by the keyword **`member`**, in which case it is related by regular membership (see 7.3.2.6 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;
}
```

The conditions that a type places on its instances (e.g., what feature it has) are always considered *necessary.* They can be indicated as *sufficient* by placing the keyword **`all`** after the keyword **`type`**. In this case, the type places additional sufficiency conditions on its instances corresponding to all the necessary conditions. 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. (See also the discussion in 7.3.2.1.)

```
type all Car specializes MaterialThing {
    feature wheels[4] : Wheel;
}
```

7.3.2.3 Specialization

Specializations are relationships between types, identified as *specific* and *general*, indicating that all instances of the specific type are instances of the *general* one (that is, 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. A type may participate in multiple specialization relationships, both as specific and general types.

A specialization relationship is declared using the keyword **`specialization`**, optionally followed by a short name and/or a name. The qualified name of the specific type, or a feature chain (see 7.3.4.6) 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`**. A specialization declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

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

```
    doc /* This specialization is unnamed. */
}
```

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

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

The *directsupertypes* of a type are all the general types in specializations for which the type is the specific type, and the *direct subtypes* of a type are all the specific types in specializations for which the type is the general type. *Indirect supertypes* include, recursively, the supertypes of the direct supertypes of a type, and similarly for *indirect subtypes.*

Specialization relationships can form cycles, which means all types in the cycle have the same instances (same extent). However, since all types are required to specialize the base type `Anything` (directly or indirectly), no cycle of valid types can be entirely closed, unless it includes the type `Anything`.

The *owned specializations* of a type are those specializations that are owned relationships of the type (see 7.2.2), for which the type is the *specific* type. An owned specialization 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;
```

A type *inherits* all visible and protected memberships of the general types of its owned specializations. *Protected* memberships are all owned and inherited memberships of the general type whose visibility declared as **`protected`** (see also 7.3.2.2 on **`protected`** visibility; 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.

```
type A specializes Base::Anything {
    feature f; // Public by default.
    protected feature g;
    private feature h;
}
type B specializes A {
    // B inherits feature memberships for
    // f and g, but not h.
}
```

7.3.2.4 Conjugation

Conjugation is a relationship between types, identified as the *original* type and the *conjugated* type, indicating the conjugated type inherits visible and protected memberships from the original type, except the direction of input and output features is reversed (see 7.3.4.1 on features with direction). Features with direction **`in`** relative to the original type are treated as having direction **`out`** relative to the conjugated type, and vice versa for direction **`out`** treated as **`in`**. Features with with no direction or direction **`inout`** in the original type are inherited without change.

A conjugation relationship is declared using the keyword **`conjugation`**, followed by a short name and/or a name. The qualified name of the conjugated type, or a feature chain (see 7.3.4.6) if the conjugated type is such a feature, is then given after the keyword **`conjugate`**, followed by the qualified name of the original type, or a feature chain if the original type is such a feature, after the keyword **`conjugates`**. The symbol ~ can be used interchangeably with the keyword **`conjugates`**. A conjugation declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
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 {
    doc /* This conjugation is equivalent to c1. */
}
```

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

```
conjugate Conjugate1 conjugates Original;
conjugate Conjugate2 ~ Original;
```

An *owned conjugation* is an owned relationship of a type (7.2.2) that is a conjugation relationship, for which the type is the *conjugated* type. An owned conjugation 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 original type after the keyword **`conjugates`** (or the symbol ~).

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

A type can be the conjugated type of at most one conjugation relationship, and a conjugated type cannot be the specific type in any specialization relationship.

7.3.2.5 Disjoining

Types related by *disjoining* do not share instances (instances cannot be in more than one of the extents; the extents are *disjoint*). 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.

A disjoining relationship is declared using the keyword **`disjoining`**, optionally followed by a short name and/or a name. The qualified name of the first type, or a feature chain (see 7.3.4.6) if the type is such a feature, is then given after the keyword **`disjoint`**, followed by the qualified name of the second type, or a feature chain, if the the type is such a feature, after the keyword **`from`**. A disjoining declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
disjoining Disj disjoint A from B;
disjoining disjoint Mammal from Mineral;
disjoining disjoint Person::parents from Person::children {
    doc /* No Person can be their own parent. */
}
```

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

```
disjoint A from B;
disjoint Mammal from Mineral;
disjoint Person::parents from Person::children;
```

An *owned disjoining* of a type is an owned relationship of the type (see 7.2.2) that is a disjoining relationship. An owned disjoining 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 disjoining type in a list after the keyword **`disjoint from`**, placed after any owned specializations.

```
type C specializes Anything disjoint from A, B;
type Mammal :> Animal disjoint from Mineral;
```

7.3.2.6 Feature Membership

A *feature membership* is a relationship between a type and a feature that is both a kind of owning membership and a kind of *type featuring* (see 7.3.4.8). Features related to a type via feature membership are identified as *owned features of the type*. The owning type is one of the feature's featuring types, meaning that the feature specifies a relation between the owning type and the type of the feature.

A feature that is declared within the body of a type is normally an owned feature of that type, so it automatically has that type as a featuring type (because feature membership is a kind of type featuring). This also applies to the bodies of classifiers (see 7.3.3) and features (see 7.3.4), since they are kinds of types. A feature may also be aliased in a type like any other Element (see 7.2.5), in which case it is related to the aliasing type by a regular membership relationship, not a feature membership, and, so, does not become one of the owned features of the type.

```
feature person[*] : Person;
classifier Person {
    // This declares an owned feature using a feature membership.
    feature age[1] : ScalarValues::Integer;

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

However, if a feature declaration in the body of type is preceded by the keyword **`member`**, then the feature is owned by the containing type via a membership relationship, not a feature membership. In this case, the feature is *not* an owned feature of the containing type, and it does *not* automatically have the containing type as a featuring type, though it may have featuring types declared in its **`featured by`** list (see 7.3.4.1 on declaring the owned typings 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;
}
```

7.3.2.7 Unioning, Intersecting, and Differencing

Unioning, intersecting, and *differencing* are relationships between an owning type and a set of other types.

1. *Unioning* specifies that the owning type classifies everything that is classified by *any* of the unioned types.
2. *Intersecting* specifies that the owning type classifies everything that is classified by *all* of the intersecting types.
3. *Differencing* specifies that the owning type classifies everything that is classified by the first of the differenced types but *not* by any of the remaining types.

Since these relationships are always owned by the source type, they are defined as part of the declaration of that type, using the keywords **`unions`**, **`intersects`**, and **`differences`**, respectively, followed by a list of qualified names (or feature chains, if appropriate, see 7.3.4.6) of the related types. These relationship clauses are placed after any owned specializations (see 7.3.2.3) but may otherwise appear in any order with each other and with any disjoining clause (see 7.3.2.5).

```
classifier Adult;
classifier Child;

classifier Person unions Adult, Child {
    feature dependents : Child[*];
    feature offspring : Person[*];
    feature grownOffspring : Adult[*] :> offspring;
    feature dependentOffspring : Child[*] :> dependents, offspring
        differences offspring, grownOffspring
        intersects dependents, offspring;
}
```

Multiple relationships of each kind can be specified using multiple clauses in a single declaration. In the case of differencing, any additional **differences** clauses after the first one mean that the owning type does *not* classify anything classified by any of the related types. It is not allowable, though, for a type to have just one of any of these relationships over all.

```
// This is valid.
classifier Person unions Adult unions Child;

// This is NOT valid.
classifier Person unions Adult;
```

7.3.3 Classifiers

7.3.3.1 Classifiers Overview

Metamodel references:

- *Concrete syntax, 8.2.4.2*
- *Abstract syntax, 8.3.3.2*
- *Semantics, 8.4.3.3*

Classifiers are types that classify things in the modeled system, as distinct from features, which model the relations between them (see 7.3.4). *Subclassification* is a kind of specialization that specifically relates classifiers.

7.3.3.2 Classifier Declaration

The notation for a classifier is the same as the generic notation for a type (see 7.3.2.2), 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* (see 7.3.3.3). A classifier is also not required to have any owned subclassifications 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` library model (see 9.2.2).

```
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 conjugated type (see 7.3.2.4), in which case the original type must also be a classifier.

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

7.3.3.3 Subclassification

A subclassification relationship is declared using the keyword **`specialization`**, optionally followed by a short name 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`**. A subclassification declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
specialization Super subclassifier A specializes B;
specialization subclassifier B :> A {
    /* This subclassification is unnamed. */
}
```

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

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

An owned subclassification 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 superclassifier in a list after the keyword **`specializes`** (or the symbol `:>`).

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

7.3.4 Features

7.3.4.1 Features Overview

Metamodel references:

- *Concrete syntax, 8.2.4.3*
- *Abstract syntax, 8.3.3.3*
- *Semantics, 8.4.3.4*

Features are types that classify how things in a modeled system are related, including by chains of relations. 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 *featuring types* (the *domain*) with instances in the intersection of the extents of its *featured types* (the *co-domain*). Instances in the domain of a feature are said to "have values" that are instances of the co-domain. The domain of features with no explicit featuring types is the type `Anything` from the `Base` library model (see 9.2.2).

Type featuring is a relationship between a feature and a type that identifies the type as a *featuring type* of the feature. *Feature membership* is both a kind of owning membership and a kind of type featuring, by which a type owns a feature and becomes a featuring type of that feature (see 7.3.2.6).

There are also several forms of specialization that apply specifically to features.

- *Feature typing* is a relationship between a feature and a type that identifies the type as a *featured type* of the feature.
- *Subsetting* is a relationship between a specific feature (the *subsetting feature*) and a more general feature (the *subsetted feature*), where the specific feature may further constrain the featuring types, featured types and multiplicity of the general feature.
- *Redefinition* is a kind of subsetting in which the specific feature (the *redefining feature*) also replaces an otherwise inherited general feature (the *redefined feature*) in the context of the owning type of the specific feature.

7.3.4.2 Feature Declaration

The notation for a feature is similar to the generic notation for a type (see 7.3.2.2), except using the keyword **feature** rather than **type**. Further, a feature can have any of three kinds of specialization, each identified by a specific keyword or equivalent symbol:

- **typed by** or `:` – Specifies FeatureTyping (see 7.3.4.3).
- **subsets** or `:>` – Specifies Subsetting (see 7.3.4.4).
- **redefines** or `:>>` – Specifies Redefinition (see 7.3.4.5).

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` library model (see 9.2.2). This is true even if a feature typing 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 conjugated type (see 7.3.2.4), in which case the original type must also be a feature. In this case, the feature must not have any owned specializations.

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

As for any type, the multiplicity of a feature can be given in square brackets `[...]` after any identification of the feature (see also 7.3.2.2). 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 7.3.4.5).

```
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). The properties flagged by these keywords are only meaningful if the feature has a multiplicity upper bound greater than one.

- **nonunique** – If a feature is *non-unique,* then, for any domain instance, the same co-domain instance may appear more than once as a value of the feature. The default is that the feature is *unique.*
- **ordered** – If a feature is *ordered,* then for any domain instance, the values of the feature can be placed in order, indexed from 1 to the number of values. The default is that the feature is *unordered.*

```
feature sensorReadings : ScalarValues::Real [*] nonunique ordered;
```

There are four other kinds of relationships that can be declared as owned relationships of a feature, each indicated by a specific keyword:

- **disjoint from** – Specifies disjoining (see 7.3.2.5).
- **chains** – Specifies feature chaining (see 7.3.4.6).
- **inverse of** – Specifies feature inverting (see 7.3.4.7).
- **featured by** – Specifies type featuring (see 7.3.4.7).

The clauses for these relationships must appear after any specialization or conjugation part, but can otherwise appear in any order.

```
feature cousins : Person[*] chains parents.siblings.children featured by Person;
feature children : Person[*] featured by Person inverse of parents;
```

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 the *direction* of a feature, which determines what is allowed to change its values on instances of its domain:
    - **in** – Things "outside" the instance. These features identify things input to an instance.
    - **out** – The instance itself or things "inside" it. These features identify things output by an instance.
    - **inout** – Both things "outside" and "inside" the instance. These features identify things that are both input to and output by an instance.
2. **abstract** – Specifies that the feature is *abstract* (see 7.3.2.2 on abstract types in general).
3. **composite** or **portion** – Specifies that the feature is either a *composite* or *portion* feature (specifying both is not allowed).
    - Values of a composite feature, on each instance of the feature's domain, cannot exist after the featuring instance ceases to exist. This only applies to values at the time the instance goes out of existence, not to other things in the co-domain that might have been values before that.
    - Portion features are composite features where the values cannot exist without the whole, because they are the “same thing” as the whole. (For example, the portion of a person's life when they are a child cannot be added or removed from that person's life.)
4. **readonly** – Specifies that the feature is *read only.* Values of read only features on each instance of their domain are the same during the entire existence of that instance.
5. **derived** – Specifies that the feature is *derived.* Such a feature is typically expected to have a bound feature value expression that completely determines its value at all times (see 7.4.11 on feature values, which is a kernel concept).
6. **end** – Specifies that the feature is an *end feature.* Any kind of type can have end features, but they are mostly used in associations (see 7.4.5) and connectors (see 7.4.6), and they are further described in those contexts.

(Note that the semantics of **composite**, **portion**, and **readonly** require a model of things existing in time, which is provided in the Kernel layer, see 7.4.3).

```
classifier Fuel {
    portion feature fuelPortion : Fuel;
}

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

assoc VehicleRegistration {
```

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

7.3.4.3 Feature Typing

A feature typing relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the typed feature is then given after the keyword **typing**, followed by the qualified name of the type, or a feature chain (see 7.3.4.6), after the keyword **typed by**. The symbol **:** can be used interchangeably with the keyword **typed by**. A feature typing declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
specialization t1 typing customer typed by Person;
specialization t2 typing employer : Organization {
    doc /* An employer is an Organization. */
}
```

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

```
typing customer typed by Person;
typing employer : Organization;
```

An *owned feature typing* is a feature typing that is an owned relationship of its type feature. An owned feature typing is defined as part of the declaration of the typed feature, rather than in a separate declaration, by including the qualified name or feature chain for the type in a list after the keyword **typed by** (or the symbol **:**).

```
feature foodItem typed by Food, InventoryItem;
```

7.3.4.4 Subsetting

Subsetting is a kind of specialization between two features. This means that the values of the subsetting feature are also values of the subsetted feature on each instance (separately) of the domain of the subsetting feature.

A subsetting relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the subsetting feature, or a feature chain (see 7.3.4.6), is then given after the keyword **subset**, followed by the qualified name of the subsetted feature, or a feature chain, after the keyword **subsets**. The symbol **:>** can be used interchangeably with the keyword **subsets**. A subsetting declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
specialization Sub subset parent subsets person;
specialization subset mother subsets parent {
    doc /* All mothers are parents. */
}
```

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

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

An *owned subsetting* is a subsetting that is an owned relationship of the subsetting feature. An owned subsetting is defined as part of the declaration of the subsetting feature, rather than in a separate declaration, by including the qualified name or feature chain of the subsetted feature in a list after the keyword **subsets** (or the symbol **:>**).

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

A subsetting feature can restrict aspects of the subsetted feature, otherwise it will, by default, have the same properties as the subsetted feature. In particular, a subsetting feature can constrain its featured types to be specializations of those of the subsetted feature and add additional feature types. A subsetting feature can also restrict the multiplicity of its subsetted feature to allow cardinalities that are smaller than those of the subsetted feature (e.g., by specifying smaller lower and/or upper bounds).

```
classifier Wheel;
classifier DriveWheel specializes Wheel;
feature anyWheels[*] : Wheel;

classifier Automobile {
    // Restricts multiplicity
    composite feature wheels[4] subsets anyWheels;
    // Restricts multiplicity and type.
    composite feature driveWheels[2] : DriveWheel subsets wheels;
}
```

If a subsetted feature is ordered, then the subsetting feature must also be ordered. If the subsetted feature is unordered, then the subsetting feature will be unordered by default, unless explicitly flagged as **ordered**.

```
classifier Automobile {
    composite feature wheels[4] ordered subsets anyWheels;
    // driveWheels must be ordered because wheels is ordered.
    composite feature driveWheels[2] ordered : DriveWheel subsets wheels;
}
```

If a subsetted feature is unique, then the subsetting feature must not be specified as non-unique. If the subsetted feature is non-unique, then the subsetting feature 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 subsets accessibleURLs; // Cannot be nonunique.
}
```

7.3.4.5 Redefinition

Redefinition is a kind of subsetting that requires the values of the redefining feature and the redefined feature to be the same on each instance (separately) of the domain of the redefining feature. This means any restrictions on the values of the redefining feature relative to the redefined feature, such as typing or multiplicity, also apply to the values of the redefined feature, and vice versa.

A redefinition relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the redefining feature, or a feature chain (see 7.3.4.6), is then given after the keyword **redefinition**, followed by the qualified name of the redefined feature, or a feature chain, after the keyword **redefines**. The symbol `:>>` can be used interchangeably with the keyword **redefines**. A redefinition declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
specialization Redef redefinition LegalRecord::guardian redefines parent;
specialization redefinition Vehicle::vin redefines RegisteredAsset::identifier {
    doc /* A "vin" is a Vehicle Identification Number. */
}
```

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

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

A feature can only be redefined once for any featuring type. A feature without any feature types is considered to be implicitly featured by the most general base type `Anything` (see 7.3.4.1). It is therefore allowable to redefine such a feature by a redefining feature that does have some other featuring type. It is, however, illegal for one such feature to redefine another, because that would correspond to a semantically inconsistent redefinition of one feature of `Anything` by another.

The restrictions on the specification of the multiplicity, ordering and uniqueness of a subsetting feature (see 7.3.4.4) also apply to a redefining feature. In addition, the multiplicity of a redefining feature must only allow cardinalities that are consistent with the multiplicity of the redefined feature (e.g., it cannot have a multiplicity lower bound that is less than that of the redefined feature).

An *owned redefinition* is a redefinition that is an owned relationship of its redefining feature. An owned redefinition 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 redefined feature in a list after the keyword **`redefines`** (or the symbol `:>>`).

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

If a redefining feature is declared as an owned feature of a type (see 7.3.2.6), then each of the redefined features of its owned redefinitions must be features that would otherwise be inherited from supertypes of its owning type. When redefined, however, these otherwise inheritable features are *not* inherited and are, instead, replaced by the redefining feature. This enables the redefining feature to have the same name as a redefined feature, if desired. (Note, however, that even though a redefined feature is not in the namespace of the owning type of the redefining feature, the redefined feature still has values on instances of that type, particularly when they are considered as instances of the supertype that owns the redefined feature. The values will be the same as for the redefining feature, as described above.)

In general, the resolution of a qualified name begins with the namespace in which the name appears and proceeds outwards from there to containing namespaces (see 8.2.3.5). However, the resolution of the qualified names of redefined features of owned redefinitions follow special rules. In particular, the local namespace of the owning type of the redefining feature is *not* included in the name resolution of the redefined features, with resolution beginning instead with the direct supertypes of the owning type. Since redefined features 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. The special rules for redefined features, however, allow such a reference, because the name resolution begins with the namespaces of the supertypes of the owning type, one of which must contain the redefined feature.

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

If neither a name nor a short name is given in the declaration of a feature with an owned redefinition, then the feature is implicitly given the same name and short name as the first redefined feature (which may itself have implicit names, if the redefined feature is itself a redefining feature). These implicit names are used in name resolution, just as explicitly declared names would be. This is useful when declaring a feature that that redefines another feature in order to constrain it, while maintaining the same naming.

```
classifier WheeledVehicle {
    // The declared name is "wheels".
    composite feature wheels[1..*] : Wheel;
```

```
}
classifier MotorizedVehicle specializes WheeledVehicle {
    // The effective name is "wheels", the same name as
    // WheeledVehicle::wheels, which is being redefined.
    composite feature redefines wheels[2..4];
}
classifier Automobile specializes MotorizedVehicle {
    // The effecive name is "wheels", the same (effective) name
    // as "MotorizedVehicle::wheels", which is being redefined.
    composite feature redefines wheels[4] : AutomobileWheel;
}
```

7.3.4.6 Feature Chaining

Feature chaining is an owned relationship between the owning *chained feature* and a *chaining feature.* If a feature has any chaining features, then it must have at least two. The list of chaining features of a chained feature is called its *feature chain.*

The meaning of a chained feature depends on its feature chain. The values of a chained feature are the same as the values of the last feature in the chain. These can be found by starting with the values of the first feature (for each instance of the chained feature's domain), then on each of those, finding the values of the second feature in the chain, and so on, to values of the last feature. If a chained feature is ordered, any ordering of values earlier in the chain are imposed on values found later in the chain. If a chained feature is non-unique, duplicate values found in the last feature of the chain (which might be due to multiple values of the earlier features) are preserved in the chained feature, otherwise the last feature can have no duplicates.

A feature chain is notated as a sequence of two or more qualified names separated by dot (`.`) symbols. Each qualified name in a feature chain must resolve to a feature. The first qualified name in a feature chain is resolved in the local namespace as usual (see 8.2.3.5). Subsequent qualified names are then resolved using the previously resolved feature as the context namespace (but considering only visible memberships). This notation specifies a list of chaining features, as given by the resolution of the qualified names in the chain, in order.

The feature chain 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).

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

The featuring types of the chaining feature are implicitly considered to include the featuring types of the first chaining feature. Similarly, the featured types of the chaining feature are implicitly considered to include the featured types of the last chaining feature.

The feature chain notation may also be used to specify a related element in the declaration of any of the following relationships:

1. Specialization (see 7.3.2.3)
2. Unioning, intersecting and differencing (see 7.3.2.7)
3. Disjoining (see 7.3.2.5)
4. Subsetting (see 7.3.4.4)
5. Redefinition (see 7.3.4.5)
6. Feature inverting (see 7.3.4.7)
7. Connector (see 7.4.6, in the Kernel layer)

In this case, the related element specified using the feature chain notation becomes an owned related feature of the relationship with the feature chain as notated.

```
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 Kernel-layer concept of a feature chain expression (see 7.4.9.3). However, it is always syntactically unambiguous as to whether the notation should be parsed as a plain feature chain or as a feature chain expression.

7.3.4.7 Feature Inverting

Feature inverting is a relationship between two features whose interpretations as relations are the inverse of each other. For example, a feature identifying each person's parents is the inverse of a feature identifying each person's children. A person identified as a parent of another will identify that other as one of their children.

A feature inverting relationship is declared using the keyword **`inverting`**, optionally followed by a short name and/or a name. The qualified name of the first feature, or a feature chain (see 7.3.4.6), is then given after the keyword **`inverse`**, followed by the qualified name of the second feature, or a feature chain, after the keyword **`of`**. A feature inverting declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
inverting parent_child inverse Person::parent of Person::child {
    doc /* A Person is the parent of their children. */
}
```

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

```
inverse Person::parents of Person::children;
```

An *owned feature inverting* is a feature inverting that is an owned relationship of its first feature. An owned feature inverting is defined as part of the declaration of the inverted feature, rather than in a separate declaration, by giving the qualified name or feature chain of the other feature after the keyword **`inverse of`**.

```
classifier Person {
    feature children : Person[*];
    feature parents : Person[*] inverse of children;
}
```

Note that only a single feature identification is allowed after **`inverse of`**. While it is possible to declare multiple feature inverting relationships for a single feature, this is generally not useful.

Inverse features can be arbitrarily nested. However, while it is allowable to use feature chains in the declaration of a feature inverting relationship, note that a feature chain is a separate feature from any of the features it chains. In order to indicate that two declared features are inverses, one should use qualified names rather than feature chains.

```
classifier A {
    feature b1: B {
        feature c1: C;
    }
}
classifier C {
    feature b2: B {
        feature a2: A inverse of A::b1::c1;
    }
}
```

7.3.4.8 Type Featuring

Type featuring is a relationship between a feature and a type, identifying the type as a featuring type of the feature (see also 7.3.4.1). Feature membership is a kind of type featuring that also makes the feature an owned member of the featuring type (see 7.3.2.6).

A type featuring relationship is declared using the keyword **featuring**, optionally followed by a short name and/or a name, and the keyword **of**. The qualified name of the featured feature is then given, followed by the qualified name of the featuring type after the keyword **featured by**. A type featuring declaration can also optionally have a relationship body (see 7.2.2.3) for, e.g., nested annotations.

```
featuring engine_by_Vehicle of engine featured by Vehicle;
featuring power featured by engine {
    doc /* The engine of a Vehicle has power. */
}
```

An *owned type featuring* is a type featuring that is an owned relationship of the featured feature. An owned type featuring is defined as part of the declaration of the feature, rather than in a separate declaration, by including the qualified name of the featuring type in a list after the keyword **featured by**.

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

Note that the domain of a feature is given by the *intersection* of its featuring types. That is, in the above example, an instance in the domain of `engine` must be *both* a `Vehicle` *and* a `PoweredComponent`.

7.4 Kernel

7.4.1 Kernel Overview

The Kernel layer completes KerML. It extends the Core layer to add modeling capabilities beyond basic classification. These include specialized classifiers for things that have the semantics of data values (*data types*) from others that have an independent existence over time and space (*classes*), and for reified relationships between things (*associations*).

Classes have instances that exist or happen in time and space. They are divided into those for *structure* and *behavior*. 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). *Functions* are behaviors that yield a single result, which can be used to form trees of *expressions.* Interactions combine behaviors and associations. Some associations are also structures.

The Kernel layer adds semantics beyond the Core primarily by specifying how model elements use the Kernel model library (see Clause 9), rather than be specified mathematically as in the Core. In the simplest case, The Kernel textual syntax introduces keywords that translate to patterns of using Core abstract syntax and library models, acting as syntactic "markers" for modeling patterns tying Kernel to the Core. In the simplest case, this involves introducing implicit specializations of model library types. For example, classes must directly or indirectly subclassify the library class `Object`, while behaviors must directly or indirectly sub classify the library class `Performance`. 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`.

This is also how other modeling languages can be built on KerML. Domain-specific metamodels and libraries can also reuse 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 Data Types

Metamodel references:

- *Concrete syntax,* 8.2.5.1
- *Abstract syntax,* 8.3.4.1
- *Semantics,* 8.4.4.2

Data types are classifiers that classify *data values* (see 9.2.2.2.2). Certain *primitive* data types have have specified extents of values, such as the numerical and other types from the `ScalarValues` library model (see 9.3.2). Other data types have features whose values can distinguish one instance of the data type from another. But, otherwise, different data values are not distinguishable.

This means that data types cannot also be classes or associations, or share instances with them. It also means that data types classify things that do not exist in time or space, because they require changing relations to other things. The feature values of a data value cannot change over time, because different feature values would inherently identify a different data value.

A data type is declared as a classifier (see 7.3.3), using the keyword **`datatype`**. If no owned superclassing is explicitly given for the data type, then it is implicitly given a default superclassing to the data type `DataValue` from the `Base` library model (see 9.2.2).

If any of the types of a feature are data types, then all of them must be. If a feature has data types as its types, and no owned subsetting or owned redefinition 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 9.2.2).

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

7.4.3 Classes

Metamodel references:

- *Concrete syntax,* 8.2.5.2
- *Abstract syntax,* 8.3.4.2
- *Semantics,* 8.4.4.3

Classes are classifiers that classify *occurrences*, which exist in time and space (see 9.2.4). Relations between an occurrence and other things can change over time and space, while the occurrence still maintains an independent identity.

A class is declared as a classifier (see 7.3.3), using the keyword **`class`**. If no owned superclassing is explicitly given for the class, then it is implicitly given a default superclassing to the class `Occurrence` from the `Occurrences` model library (see 9.2.4).

If any of the types of a feature are classes, then all of them must be. If a feature has class types, and no owned subsetting or owned redefinition is explicitly given in the feature declaration, then the feature is implicitly given a default subsetting to the feature `occurrences` from the `Occurrences` library model (see 9.2.4), unless at least one of the types is an association structure, in which case the default subclassing is as described in 7.4.5.

```
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.
}
```

7.4.4 Structures

Metamodel references:

- *Concrete syntax,* 8.2.5.3
- *Abstract syntax,* 8.3.4.3
- *Semantics,* 8.4.4.4

Structures are classes that classify o*bjects*, which are kinds of occurrences. Structures typically limit how their instances and relations between them can change over time, as opposed to Behaviors, which 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 behaviors.

A structure is declared as a classifier (see 7.3.3), using the keyword **`struct`**. If no owned superclassing is explicitly given for the structure, then it is implicitly given a default superclassing to the structure `Object` from the `Objects` library model (see 9.2.5).

If any of the types of a feature are structures, then all of them must be. If a feature has structure types, and no owned subsetting or owned redefinition is explicitly given in the feature declaration, then the feature is implicitly given a default subsetting to the feature `objects` from the `Objects` library model (see 9.2.5), unless at least one of the types is an association structure, in which case the default subsetting shall be as specified in 7.4.5.

```
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.
}
```

7.4.5 Associations

Metamodel references:

- *Concrete syntax,* 8.2.5.4
- *Abstract syntax,* 8.3.4.4
- *Semantics,* 8.4.4.5

Associations are classifiers that classify *links* between things (see 9.2.3.1) At least two owned features of an association must be end features (see 7.3.4.2), its *association ends*, 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 association ends are called *binary associations*. Associations can also have features that are not end features, which characterize each instance of the association separately from the things it links.

An association is declared as a classifier (see 7.3.3), using the keyword **assoc**. If no owned superclassing 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 9.2.3).

An association is also a relationship between the types of its association ends, which might be the same type, and are identified by its *related types*. Links are between instances of an association's related types. For binary associations, the two related types are identified as the *source type* and the *target type*, which might be the same. Associations with more than two association ends ("n-ary") have only target types, no source types.

The semantics of multiplicity is different for end features from that for non-end features (as described in 7.3.4.2). The end features of an association determine the *participants* in the links that are instances of the association and, as such, effectively have multiplicity of "1" relative to the association. But, if an association end has a multiplicity specified other than `1..1,` then this is interpreted as follows: For each association end, the multiplicity, ordering and uniqueness constraints specified for that end apply to each set of instance of the association that have the same (single) values for each of the other ends. For a binary association, this corresponds to the multiplicity resulting from "navigating" across the association given a value at one end of the association to the other end of the association. (See also 8.4.4.5 on association semantics.)

If an association has a single superclassifier that is an association, it may inherit association ends from this superclassifier association. However, if it declares any owned association ends, then each of these must redefine an association end of the superclassifier association, in order, up to the number of association ends of the superclassifier. If no redefinition is given explicitly for an owned association end, then it is considered to implicitly redefine the association end at the same position, in order, of the superclassifier Association (including implicit defaults), if any.

```
assoc Ownership { // Specializes Links::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 superclassifier that is an association, then the association *must* declare a number of owned association ends at least equal to the maximum number of association ends of any of its superclassifier associations. Each of these owned association ends must then redefine the corresponding association end (if any) at the same position, in order, of each of the superclassifier associations.

Association structures are both associations and structures (see 7.4.4 on structures), classifying *link objects,* which are both links and objects (see 9.2.5.1 on objects). As objects, link objects can be created and destroyed, and their non-end features can change over time. However, the values of the end features of a link object are fixed and cannot change over its lifetime (that is, they are effectively *read only,* see also 7.3.4.2).

An association structure is declared like a regular association, but using the keyword **assoc struct**. An association structure must directly or indirectly specialize the base associations structure `LinkObject`. If this is not the case due to the explicit owned superclassifications in its declaration, then it is implicitly given a default superclassing to either the association structure `BinaryLinkObject` (if it is a binary association structure) or the association structure `LinkObject` (otherwise), both of which are from the `Objects` library model (see 9.2.5). The same rules on association ends described above for associations also apply to association structures. An association structure may specialize an association that is not an association structure, but all subclassifications of an association structure must be association structures.

```
// Also implicitly specializations Objects::BinaryLinkObject.
assoc struct ExtendedOwnership specializes Ownership {
    // End features are inherited from Ownership.
    // The values of the feature "revaluations" may change over time.
    feature revaluations[*] ordered : MonetaryValue;
}
```

If a feature has one or more associations as types, then these associations must all have the same number of association ends. If the feature defines owned end features in its body, then it can have no more than the number of association ends of its association types. The owned end features of such a feature follow the same rules for redefinition of the association ends of its association types as described above for the redefinition of the association ends of superclassifier associations by a subclassifier association.

If a feature declaration has no explicit owned subsettings or owned redefinitions, and any of its types are binary associations, then the feature is implicitly given a default subsetting to the feature `binaryLinks` from the `Links` library model (see 9.2.3) or to the feature `binaryLinkObjects` from the `Objects` library model (see 9.2.5), if any of the associations are association structures. 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` library model (see 9.2.3) or to the feature `linkObjects` from the `Objects` library model (see 9.2.5), if any of the associations are association structures.

7.4.6 Connectors

7.4.6.1 Connectors Overview

Metamodel references:

- *Concrete syntax,* 8.2.5.5
- *Abstract syntax,* 8.3.4.5
- *Semantics,* 8.4.4.6

Connectors are features that are typed by associations (see 7.4.5), having values that are links (see 9.2.3.1). Like an association, a connector has end features, known as its *connector ends.* Each connector end redefines an association end from each of the associations that type the connector and subsets a feature that becomes a *related feature* of the connector. Connectors typed by binary associations are called *binary connectors.*

A connector is also a relationship between its related features. For binary connectors, the two related features are identified as the *source feature* and the *target feature,* which might be the same. Connectors with more than two connector ends ("n-ary") have only target features, no source features.

Connectors can be thought of as "instance-specific" associations, because their values (which are links) are each limited to linking things identified via related features on the same instance of the connector's domain (or by things identified by that instance, recursively, see below). For example, an association could be used to model an engine driving wheels, and to type a connector in the car model. 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 association ends, regardless of the domain of the connector
2. identified by its related features for the same instance of the domain of the connector (or by things identified by that instance, recursively).

For example, if the wheels in a car are taken to be part of its drive train, rather than part of the car directly, then the engine in each car will drive wheels identified by that car's drive train, rather than a feature of the car directly. This requires that each related feature of a connector have some featuring type of the connector as a direct or indirect featuring type (where a feature with no featuring type is treated as if the classifier `Anything` was its featuring type). In particular, this condition is satisfied if a connector has an owned type that either also directly owns the related features of the connector or from which the related features can be reached by chaining (see 7.3.4.6). Otherwise, explicit owned type featurings (see 7.3.4.8) should be used to ensure that the connector has a sufficiently general domain.

Binding connectors are binary connectors that require their source and target features to have the same values on each instance of their domain. They are typed by the library association `SelfLink` (which only links things in the modeled universe to themselves, see 9.2.3.1) and have end multiplicities of exactly 1. This requires a `SelfLink` to exist between each value of the source feature and exactly one value of the target feature, and vice-versa.

To be meaningful, the declared co-domains of the related features of a binding connector must at least overlap. Since the interpretations of data types are disjoint from those of classes, this means that a feature typed by data types can only be bound to another feature typed by data types. In the determination of the equivalence of such features, indistinguishable data values are considered equivalent. The binding of features typed by classes to another feature typed by classes, on the other hand indicates that the same occurrences play the roles represented by each of the related features.

Successions are binary connectors requiring their source and target features to identify *Occurrences* that are ordered in time. They are typed by the library association `HappensBefore` (see 9.2.4.1), which links occurrences that happen completely separately in time, with the connector's source feature being the earlier occurrence and the target feature being the later occurrence.

7.4.6.2 Connector Declaration

A connector is declared as a feature (see 7.3.4.2) using the keyword **`connector`**. If no owned subsetting or owned redefinition is explicitly given for a connector, and none of its types are association structures, then the connector is implicitly given a default subsetting to the feature `binaryLinks` from the `Links` library model (see 9.2.3), if it is a binary connector, or to the feature `links` from the `Links` library model, otherwise. If at least one of the types of a connector is an association structure, then the default subsetting is instead to the feature `binaryLinkObjects` from the `Objects` library model (see 9.2.5), if it is a binary connector, or to the feature `linkObjects` from the `Objects` library model, otherwise.

In addition, a connector declaration includes *connector end* features that redefine the ends of the associations that type the connector and reference the features related by the connector (see also the description of association ends in 7.4.5). All associations typing a connector must have the same number of association ends, which are the same as the number of related features of the connector. The connector ends are declared in the same order as the corresponding association ends. The related feature referenced by a connector end is specified using the keyword **`references`** or the equivalent symbol `::>`.

```
// Specializes Objects::BinaryLinkObject 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::binaryLinkObjects by default.
    connector mount[2] : Mounting {
        end feature mountingAxle references axle;
```

```
        end feature mountedWheel references wheels;
    }
}
```

The **references** notation indicates that connector end features have *reference subsetting* relationships to the features related by the connector. Reference subsetting has the same semantics as regular subsetting (see 7.3.4.4) but is used to syntactically differentiate one of the owned subsettings of a feature. While reference subsetting is used primarily for connector ends in KerML, it can actually be specified as an owned subsetting in the declaration of any kind of feature, using the **references** or `::>` symbol. A feature is allowed to have at most one owned subsetting that is a reference subsetting.

```
struct WheelAssembly {
    composite feature axle[1] : Axle {
        feature mountedWheels[2] : Wheel;
    }
    composite feature wheels[2] : Wheel;
    connector mount[2] : Mounting {
        end mountingAxle references axle;
        end mountedWheel references wheels subsets mountingAxle.mountedWheels;
    }
}
```

Instead of explicitly declaring connector ends in the body of the connector, they can be listed between parentheses, after the regular feature declaration part and before the body of the connector (if any). In this case, the end declarations are limited to be of the form *`e`* **`references`** *`f`* or *`e`* `::>` *`f`*, where *`e`* is the name of an association end and *`f`* is the qualified name of a related feature.

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

The association end names can also be omitted, in which case the connector ends are matched in order to corresponding association ends.

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

    connector mount[2] : Mounting (axle, wheels);
}
```

By default, the connector ends of a connector are declared in the same order as the association ends of the types of the connector. However, if the connector has a single type, then the related features can be given in any order, with each related feature paired with an association end of the type using a notation of the form *`e`* **`references`** *`f`* or *`e`* `::>` *`f`*, where *`e`* is the name of an association end and *`f`* is the qualified name of a related feature. In this case, the name of each association end must appear exactly once in the list of connector end declarations.

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

```
struct WheelAssembly {
    composite feature axle[1] : Axle;
```

```
    composite feature wheels[2] : Wheel;

    connector mount : Mounting from axle to wheels;
}
```

If a binary connector declaration includes only the related features part, then the keyword `from` can be omitted.

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

    // Subsets Links::binaryLinks by default.
    connector axle to wheels;
}
```

If a binary connector has a single type, then the names of the association ends of the type can also be used in the declaration of the connector ends in the special notation for binary connectors. However, since the connector ends are always declared in order from source to target in this notation, the association end 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 connector end, after the qualified name of the related feature for that end. In this case, the given multiplicity redefines the multiplicity that would otherwise be inherited from the association end corresponding to the connector end.

```
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 owned feature of a type (as above), the context consistency condition for the related features of the connector (see 7.4.6.1) requires that these features also be directly or indirectly nested within the owning type. The feature chain dot notation (see 7.3.4.6) should be used when connecting so-called "deeply nested" features.

While the resolution of a feature chain 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 chain *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;
}
```

7.4.6.3 Binding Connector Declaration

A binding connector is declared as a feature (see 7.3.4.2) using the keyword **binding**. In addition, a binding connector declaration gives, after the keyword **of**, the qualified names of the two related features that are bound by the binding connector, separated by the symbol =, after the regular feature declaration part and before the body of the binding connector (if any). If no owned subsetting or owned redefinition is explicitly given, then the binding connector is implicitly given a default subsetting to the feature `selfLinks` from the `Links` library model (see 9.2.3). Note that, due to this default subsetting, if no `type` is explicitly given for a binding connector, then it will implicitly have the `type SelfLink` (the type of `selfLinks`).

```
struct Vehicle {
    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 binding connector declaration includes only the related features part, then the keyword **of** can be omitted.

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

    composite feature engine {
        in feature fuelFlowIn : Fuel;
    }

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

The connector ends of a binding connector always have multiplicity `1..1`.

(See also 7.4.11 on the use of binding connectors with feature values.)

7.4.6.4 Succession Declaration

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 feature after the keyword **first** and the qualified name of the target feature after the keyword **then**. If no owned subsetting or owned redefinition is explicitly given, then the succession is implicitly given a default subsetting to the feature `happensBeforeLinks` from the `Occurrences` library model (see 9.2.4). Note that, due to this default subsetting, if no `type` is explicitly given for a succession, then it will implicitly have the `type HappensBefore` (the type of `happensBeforeLinks`).

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

If a succession declaration includes only the related features part, then the keyword **`first`** can be omitted.

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

As for connector ends on regular connectors, constraining multiplicities can also be defined for the connector ends 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];
}
```

7.4.7 Behaviors

7.4.7.1 Behaviors Overview

Metamodel references:

- *Concrete syntax,* 8.2.5.6
- *Abstract syntax,* 8.3.4.6
- *Semantics,* 8.4.4.7

Behaviors are classes that classify *performances*, which are kinds of occurrences that can be spread out in disconnected portions of space and time (see 9.2.6). The performance of behaviors can cause effects on other things, including their existence and relations, some of which might be accepted as input to or provided as output from the behavior.

Behaviors can have *steps,* which are features typed by behaviors, allowing the containing behavior to coordinate the performance of other behaviors. Steps can be ordered in time using succession connectors (see 7.4.6.4). They can also be connected by item flows to model things flowing between the output of one step and the input of another. Steps can also nest other steps to augment or redefine steps inherited from their behavior types.

7.4.7.2 Behavior Declaration

A behavior is declared as a classifier (see 7.3.3), using the keyword **`behavior`**. If no owned superclassing is explicitly given for the behavior, then it is implicitly given a default superclassing to the behavior `Performance` from the `Performances` library model (see 9.2.6).

Features declared in the body of a behavior with a non-null direction (see 7.3.4.2) are considered to be the owned *parameters* of the behavior. Features with direction **in** are input parameters, those with direction **out** are output parameters, and those with direction **inout** are both input and output parameters.

```
// Specializes Performances::Performance by default.
behavior TakePicture {
    in scene : Scene;
    out picture : Picture;
}
```

Parameters are ordered in the lexical order they are declared in the body of a behavior. They may appear at any location within the body.

If a behavior has owned subclassifications whose superclassifiers are behaviors, then each of the owned parameters of the subclassifier behavior must, in order, redefine the parameter at the same position of each of the superclassifier behaviors. The redefining parameters shall have the same direction as the redefined parameters.

```
behavior A { in a1; out a2; }
behavior B { in b1; out b2; }
behavior C specializes A, B {
    in c1 redefines a1, b1;
    out c2 redefines a2, 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 is implicitly given owned redefinitions 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 b1; out b2; inout b3; } // Redefinitions are implicit.
behavior C1 :> A1, B1 { in c1; out c2; inout c3; }
```

Steps (see 7.4.7.3) 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 superclassifer Behavior.

```
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;
    composite step shoot : Shoot;
}
```

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;
    succession focus then shoot;
    composite flow focus.image to shoot.image;
    composite step shoot : Shoot;
```

```
        binding picture = focus.picture;
    }
```

7.4.7.3 Step Declaration

A step is declared as a feature (see 7.3.4.2) using the keyword **`step`**. If no owned subsetting or owned redefinition is explicitly given, then the step is implicitly given a default subsetting to the feature `performances` from the `Performances` library model (see 9.2.6).

As for a behavior, directed features declared in the body of a step are considered to be *parameters* of the step (see 7.4.7.2). If a step has owned specializations (including all feature typings, subsettings, and redefinitions), whose general type is a behavior or a step. then the rules for the redefinition of parameters of the behaviors and steps are the same as for the redefinition of the parameters of superclassifier behaviors by a subclassifier behavior (see 7.4.7.2).

```
step focus : Focus {
    // Parameters redefine parameters of Focus.
    in scene;
    out image;
}

// Parameters are inherited.
step refocus subsets focus;
```

A step can also have a body, which may have steps in it. A step can inherit or redefine steps from its behavior types or any other steps it subsets.

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

7.4.8 Functions

7.4.8.1 Functions Overview

Metamodel references:

- *Concrete syntax,* 8.2.5.7
- *Abstract syntax,* 8.3.4.7
- *Semantics,* 8.4.4.8

Functions are behaviors (see 7.4.7) with one out parameter designated as the *result parameter*. Functions classify *evaluations* (see 9.2.6.2.3), which are kinds of performances that produce *results* as values of the result parameter. Like all behaviors, functions can change things, often referred to as "side effects". A *pure* function is one that has no side effects and always produces the same results given the same input values, similarly to a function in the mathematical sense. The numerical functions in the Kernel Function Library (see 9.4), for example, are pure functions.

Expressions are steps (see 7.4.7) typed by only a single function, which means that their values are evaluations. An expression whose value is an evaluation with results is said to *evaluate to* those results. They can be steps in any behavior, but a function, in particular, can designate one of its expression steps as the *result expression* that gives the value of its result parameter. Expressions can have their own nested parameters, to augment or redefine those of their functions, including the result parameter. They can also own other expressions and designate a result expression, similarly to a function. (See also 7.4.9 for more on expressions).

Predicates are functions whose result is a single Boolean value (that is, true or false). A predicate determines whether the values of its input parameters meet particular conditions at the time of its evaluation, resulting in true if they do, and false otherwise. Predicates classify *boolean evaluations,* which are specialized evaluations giving a Boolean result (see 9.2.6.2.1).

Boolean expressions are expressions whose function is a predicate and, so, evaluate to a Boolean result. A boolean expression might, in general, evaluate to true at some times and false at other times. An *invariant,* though, is a boolean expression that must always evaluate to either true at all times or false at all times. By default, an invariant is asserted to always evaluate to true, while a *negated invariant* is asserted to always evaluate to false.

7.4.8.2 Function Declaration

A function is declared as a behavior (see 7.4.7.2), using the keyword **`function`**. If no owned superclassing is explicitly given for a function, then it is implicitly given a default subclassification to the function `Evaluation` from the `Performances` library model (see 9.2.6). As for a behavior, any feature declared in the body of a function with an explicit direction is considered to be a parameter of the function. 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).

```
// Specializes Performances::Evaluation by default.
function Velocity {
    in v_i : VelocityValue;
    in a : AccelerationValue;
    in dt : TimeValue;
    return v_f : VelocityValue;
}
```

If a function has owned subclassifications that are behaviors, then the rules for redefinition or inheritance of non-result parameters are the same as for a behavior (see 7.4.7.2). If some of the superclassifier behaviors are functions, then the result parameter of the subclassifier function must redefine the result parameters of the superclassifier functions. If, in this case, the result parameter of the subclassifier function has no owned redefinitions, then it is implicitly given redefinitions of the result parameter of each of the superclassifier functions.

```
abstract function Dynamics {
    in initialState : DynamicState;
    in time : TimeValue;
    return : DynamicState;
}
function VehicleDynamics specializes Dynamics {
    // Each parameter redefines the corresponding superclassifier parameter
    in initialState : VehicleState;
    in time : TimeValue;
    return : VehicleState;
}
```

The body of a function is like the body of a behavior (see 7.4.7.2), with the optional addition of the declaration of a result expression at the end. A result expression is always be written using the Expression notation described in 7.4.9, *not* using the Expression declaration notation from 7.4.8.3. The result of the result expression is implicitly bound to the result parameter of the containing function.

```
function Average {
    in scores[1..*] : Rational;
    return : Rational;

    sum(scores) / size(scores)
}
```

Note. A result expression is written *without* a final semicolon.

The result of a function can also be explicitly bound, either using a binding connector (see 7.4.6.3) or a feature value on the result parameter declaration (see 7.4.11). In this case, the body of the function should *not* include a result expression.

```
function Average {
    in scores[1..*] : Rational;
    return : Rational = sum(scores) / size(scores);
}
```

7.4.8.3 Expression Declaration

An expression can be declared as a step (see 7.4.7.3) using the keyword **`expr`** (see also 7.4.9 for more traditional expression notation). If no owned subsetting or owned redefinition is explicitly given, then the expression is implicitly given a default subsetting to the feature `evaluations` from the `Performances` library model (see 9.2.6).

As for a step, directed features declared in the body of an expression are considered to be parameters of the expression (see 7.4.7.3). If an expression has owned specializations (including all feature typings, 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 are the same as for the redefinition of the parameters of superclassifier behaviors by a subclassifier function (see 7.4.8.2).

```
expr computation : ComputeDynamics {
    // Parameters redefined parameters of ComputeDynamics.
    in state;
    in dt;
    return result;
}
expr vehicleComputation subsets computation {
    // Input parameters are inherited, result is redefined.
    return : VehicleState;
}
```

Like a function body, an expression body can also specify a result expression.

```
expr : VehicleDynamics {
    in initialState;
    in time;
    return result;

    vehicleComputation(initialState, time)
}
```

Or the result can be explicitly bound.

```
expr : Dynamics {
    in initialState;
    in time;
    return result : VehicleState =
        vehicleComputation(initialState, time);
}
```

7.4.8.4 Predicate Declaration

A predicate is declared as a function (see 7.4.8), using the keyword **predicate**. If no owned subclassification is explicitly given for a predicate, then it is implicitly given a default subclassification to the predicate `BooleanEvaluation` from the `Performances` library model (see 9.2.6). If a predicate has owned subclassifications that are behaviors, then the rules for redefinition or inheritance of non-result parameters are the same as for a function (see 7.4.8.2). Since a predicate must always return a `Boolean` result, it is not necessary to explicitly declare a result parameter for it. However, if a result parameter is declared, then it must have type `Boolean` from the `ScalarValues` library model (see 9.3.2) and multiplicity `1..1` (see 7.4.12).

```
predicate isAssembled {
    in assembly : Assembly;
    in subassemblies[*] : Assembly;
}
```

The body of a predicate is the same as a function body (see 7.4.8). If a result expression is included, then it must have a `Boolean` result.

```
predicate isFull {
    in tank : FuelTank;
    tank.fuelLevel == tank.maxFuelLevel
}
```

7.4.8.5 Boolean Expression and Invariant Declaration

A boolean expression is declared as an expression (see 7.4.8.3), using the keyword **bool**. If no owned subsetting or owned redefinition is explicitly given, then the boolean expression is implicitly given a default subsetting to the feature `booleanEvaluations` from the `Performances` library model (see 9.2.6).

As for an expression, directed features declared in the body of a boolean expression are considered to be parameters of the boolean expression (see 7.4.8.3). If a boolean expression has owned specializations (including all feature typings, 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 are the same as for a regular expression declaration (see 7.4.8.3). The requirements on, and default for, the result parameter of a boolean expression are the same as for a predicate (see 7.4.8.4).

```
// All input parameters are inherited.
bool assemblyChecks[*] : isAssembled;
```

Like a predicate body (see 7.4.8.4), a boolean expression body can specify a `Boolean` result expression.

```
class FuelTank {
    feature fuelLevel : Real;
    feature readonly maxFuelLevel : Real;
    bool isFull { fuelLevel == maxFuelLevel }
}
```

An invariant is declared like any other boolean expression, except using the keyword **inv** instead of **bool**, and, additionally, this keyword may be optionally followed by one of the keywords **true** or **false**, to indicate whether the invariant is asserted to be true or false (i.e., is negated). The default is **true**.

```
class FuelTank {
    feature fuelLevel : Real;
    feature readonly maxFuelLevel : Real;
    // The invariant is asserted true by default.
    inv { fuelLevel >= 0 & fuelLevel <= maxFuelLevel }
```

```
    // The invariant is explicitly asserted false, that is, it is negated.
    inv false { fuelLevel > maxFuelLevel }
}
```

7.4.9 Expressions

7.4.9.1 Expressions Overview

Metamodel references:

- *Concrete syntax,* 8.2.5.8
- *Abstract syntax,* 8.3.4.8
- *Semantics,* 8.4.4.9

As described in 7.4.8, expressions are steps typed by functions, and 7.4.8.3 covers the general notation for declaring an expression as a step. However, expressions are commonly organized into tree structures, with expressions as the nodes, and the input parameters of each expression bound to the result of each of its child expressions. KerML includes extensive textual notation for constructing expression trees, including traditional operator notations for functions in the Kernel Model Library (see Clause 9).

These expression notations map entirely to an abstract syntax involving just a few specialized kinds of expressions:

- The non-leaf nodes of an expression tree are *invocation expressions,* a kind of expression that specifies its input values as the results of other expressions (its *argument* expressions), one for each of the input parameters of its *invoked* function.
- The edges of the tree are binding connectors between the input parameters of an invocation expression (redefining those of its `function`) and the results of its argument expressions.
- The leaf nodes are these kinds of expressions:
  - *Feature reference expressions* evaluate to values of a referenced feature that is not part of the expression tree.
  - *Literal expressions* evaluate to the literal value of one of the primitive data types from the `ScalarValues` model library (see 9.3.2).
  - *Null expressions* evaluate to the empty set.

An expression can also be the referent of a feature reference expression in an expression tree, as above. This enables the evaluation of the referent expression to be taken as the value of the argument of an invocation, rather than passing the value of the *result* of the evaluation. As a shorthand for doing this, the concrete syntax for an expression body (as described in 7.4.8.3) can be used as a leaf node in the expression syntax tree.

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 features of metadata (see 7.4.13) and be used as element filtering conditions in packages (see 7.4.14). The expressiveness of model-level evaluable expressions is restricted to support this:

- All null expressions, literal expressions and feature reference expressions are model-level evaluable.
- An invocation expression is model-level evaluable if and only if it meets the following conditions:

1. All its argument expressions are model-level evaluable.
2. It invokes a function that is listed as being model-level evaluable in Table 5 (in 8.2.5.8.1) or Table 7 (in 8.2.5.8.2).

7.4.9.2 Operator Expressions

Operator expression notation provides a shorthand for invocation expressions that invoke a library function represented as an *operator symbol.* (Table 5 in 8.2.5.8.1 shows the mapping from operator symbols to the functions they represent from the Kernel Model Library.) An operator expression contains subexpressions called its *operands* that generally correspond to the argument expressions of the invocation expression, except in the case of operators representing *control functions*, in which case the evaluation of certain operands is as determined by the function.

Operator expressions include the following:

- *Conditional expressions.* The *conditional test* operator **`if`** is followed by three operands, with the symbol `?` after the first operand and the keyword **`else`** after the second operand. A conditional expression 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.

```
if x >= 0? x else -x
```

- *Binary operator expressions.* A *binary operator* is one that has two operands. The binary operators include numerical operators (`+`, `-`, `*`, `/`, `%`, `^`, `**`), logical operators (`&`, `|`, **`xor`**), comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`, `===`, `!==`), and the range construction operator (`..`). In general, both operands become arguments of the invocation expression, with their results being passed to the invocation of the function represented by the operator. However, the null-coalescing (`??`), conditional and (**`and`**), conditional or (**`or`**) and implication (**`implies`**) operators all correspond to control functions 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, false, or true, respectively).

```
x + y
list#(i) ?? default
i > 0 and sensor#(i) != null
sensor == null or sensor.reading > 0
```

  The operators `==` and `!=` apply to operands that have single values, testing whether they are equal or unequal, respectively. They also evaluate to true or false, respectively, if their operands are both null (no values). The operators `===` and `!==` apply specifically to values that are *occurrences* (see 9.2.4). They test whether two occurrences are portions (in space and/or time) of the same *life* occurrence. Informally, these operators test whether or not two occurrences have the same "identity". For data values (values that are not occurrences), `===` and `!==` are the same as `==` and `!=`.

```
currentPortion == tripPortion  // True for same trip portions
currentPortion === tripPortion // True for any two portions of same trip
```

- *Unary operator expressions.* A *unary operator* is one that has a single operand. The result of evaluating the operand is passed to the invocation of the Function represented by the operator. The unary operators include the numerical operators `+` and `-` and the logical operator **`not`**.

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

- *Classification expressions.* 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 all of the values of their first operand is classified by the named type (either including or not including subtypes, respectively). The `@` operator is similar to **`istype`**, but tests whether at least one of the values of its first operand is classified by the named type. Note that this means that **`istype`** and **`hastype`** evaluate to true on a **`null`** (empty list) value, while `@` evaluates to false.

```
sensors istype ThermalSensor // Are all sensors ThermalSensors?
sensors @ ThermalSensor // Is any sensor a 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 guaranteed to be instances of the named type.

```
sensors 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 9.2.2.2.1). This is useful, in particular, when used as a test within an element filter condition expression (see 7.4.14).

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

- *Metaclassification expressions.* The metaclassification operators `@@` and **meta** take the qualified name of any kind of element as their first operand and a metaclass (see 7.4.13) as their second operand. They are shorthands for classification expressions with the operators `@` and **as**, respectively, and a metadata access expression (see 7.4.9.4) as their first operand. As such, `@@` tests whether any metadata associated with an element are classified by the given metaclass, while **meta** filters the metadata associated with an element and evaluates to those that are classified by the given metaclass.

```
// Shorthand for designModel.metadata @ ApprovalAnnotation
designModel @@ ApprovalAnnotation

// Shorthand for sensors.metadata as KerML::Feature
sensors meta KerML::Feature

// Evaluates to the string "sensors".
(sensors meta KerML::Feature).name
```

- *Extent expressions.* 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
```

In an operator expression containing nested operator expressions, the nested expressions are implicitly grouped according to the *precedence* of the operators involved, as given in Table 6 (in 8.2.5.8.1). Operator expressions with higher precedence operators are grouped more tightly than those with lower precedence operators. For example, the operator expression

```
-x + y * z
```

is considered equivalent to

```
( (-x) + (y * z) )
```

7.4.9.3 Primary Expressions

Primary expression notation provides additional shorthands for certain kinds of invocation expressions.For those cases in which the invoked function is represented by an operator symbol, the symbol is mapped to the appropriate library function as given in Table 7 (in 8.2.5.8.2).

Primary expressions include the following:

- *Index expression.* An index expression specifies the invocation of the indexing function `'#'` from the `BaseFunctions` library model (see 9.4.2). 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. Note that parentheses are required around the second operand.

  ```
  sensors#(activeSensorIndex)
  ```

  However, the behavior of the `'#'` operator is specialized for the `OrderedCollection` (see 9.3.3.2.7) and `Array` (see 9.3.3.2.1) data types from the `Collections` library model. In this case, the first operand must be a single value of one of these data types. For an `Array`, the second operand is a *sequence* of indexes whose size is the `rank` of the `Array` (i.e., the number of `dimensions` of the `Array`).

  ```
  detectorArray#(n, m)
  ```

- *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 9.4.2). 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 null expression (see 7.4.9.4).

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

  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 data types from the `Collections` library model (see 9.3.3).
- *Feature chain expression.* A feature chain expression consists of a primary expression and a feature qualified name or a feature chain (7.3.4.6), 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 8.2.3.5), but considering only visible 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 cannot 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 7.4.9.4) 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 7.4.9.4) 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 invocation expression 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 7.4.9.4). 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 7.4.9.4) 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 9.4.17).

  ```
  sensors -> select {in s: Sensor; s::isActive}
  members -> reject {in member: Member; not 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 expression may be specified using simply the name of the invoked function.

  ```
  factors -> reduce RealFunctions::'*'
  ```

7.4.9.4 Base Expressions

Base expression notation includes representations for literal expressions, null expressions, invocation expressions, feature reference expressions (including using expression bodies as base expressions).

- *Literal expressions* are described in 7.4.9.5.
- A *null expression* is notated by the keyword **null**. A null expression always evaluates to a result of "no values", which is equivalent to the empty sequence `()`.
- An *invocation expression* 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 `()` and separated by commas. The parentheses must be included, even if the argument list is empty.

```
IntegerFunctions::'+'(i, j)
isInGoodStanding(member#(n))
AddMember(org, member)
```

The arguments are matched to the input parameters of the given function in order. Alternatively, the arguments may be matched to parameters by name, using the form *`paramName = argExpression`*, in which case they may be given in any order. Note that, if the named argument notation is used, it must be used for all arguments.

```
AddMember(newMember = member, organization = org)
```

If the qualified name given for an invocation expression resolves to an expression instead of a function, then the invocation expression 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 {in x : Anything; return: Anything;}
function apply {
    in expr fn : UnaryFunction;
    in value : Anything;

    // Invokes UnaryFunction as specified by parameter fn.
    return : Anything = fn(value);
}
```

It is also possible to specify an expression to be invoked using a feature chain (see 7.3.4.6).

```
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 *construction expression* is an invocation expression in which the invocation target is a classifier that is *not* a function. In this case, the result of the construction expression is a new instance of the target classifier, with the results of the argument expressions bound to the public features of the type, in order or by name.

```
class Member {
    feature firstName : String;
    feature lastName : String;
    feature memberNumber : Integer;
    feature sponsor : Member[0..1];
}
feature thisMember = Member("Jane", "Doe", 1234, null);
feature nextMember = Member(
    firstName = "John", lastName = "Doe", sponsor = thisMember,
    memberNumber = thisMember.memberNumber + 1);
```

- A *feature reference expression* 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 {
    if x istype Integer? (x as Integer) + 1 else 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 7.4.8.3). Such *body expressions* are particularly useful when used for the second argument of a function operation expression (see 7.4.9.3).

```
feature two =
    apply({in x; if x istype Integer? (x as Integer) + 1 else 0}, 1);
feature incrementedValues =
    values -> collect {in x: Number; x + 1};
```

- A *metadata access expression* is represented by suffixing the qualified name of any kind of element with the notation **`.metadata`**. This is a *reflective* expression that evaluates to the sequence of metadata features associated with the named element in the model itself, as instances of their respective metaclasses (see 7.4.13 on metadata and metaclasses). In addition, the last value in the sequence is an instance of the metaclass for the named element from the *`KerML`* reflective abstract syntax model (see 9.2.17), representing its instantiation as a model element.

```
metaclass SecurityAnnotation;
class SecureSystem {
    metadata SecurityAnnotation;
}

// Two values: an instance of SecurityAnnotation
// and an instance of type KerML::Class.
feature sysMetadata = SecureSystem.metadata;
```

7.4.9.5 Literal Expressions

A *literal expression* is represented by giving a lexical literal for the value of the expression.

- A *literal Boolean* is represented by either of the keywords **`true`** or **`false`**.
- A *literal string* is represented by a lexical string value surrounded by double quotes `"…"` as specified in 8.2.2.5.

```
"This is a string literal."
```

- A *literal integer* is represented by a lexical decimal value as specified in 8.2.2.4. 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 7.4.9.2) to an unsigned decimal literal.

```
0
1234
```

- A *literal rational* is represented with a syntax constructed from lexical decimal values and exponential values (see 8.2.2.4). The full rational 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 *literal infinity* is represented by the symbol `*`.

7.4.10 Interactions

7.4.10.1 Interactions Overview

Metamodel references:

- *Concrete syntax,* 8.2.5.9
- *Abstract syntax,* 8.3.4.9
- *Semantics,* 8.4.4.10

Interactions are behaviors that are also associations (see 7.4.7 and 7.4.5, respectively), classifying performances that are also links between occurrences (see 9.2.3 through 9.2.6). They specify how the linked participants affect each other and collaborate.

Transfers are interactions between two participants that carry items from one occurrence to another, with items optionally identified by output and input features of the source and target occurrence, respectively (see 9.2.7).

Item flows are steps that are also binary connectors (see 7.4.7 and 7.4.6, respectively), with values that are transfers. An item flow optionally ensures that items are transferred from an output feature of the connected source feature to an input feature of the target feature. *Succession item flows* are item flows that are also successions (see 7.4.6). They identify transfers that happen after their source (that is, after the end of the occurrence where the items come from) and before their target (that is, before the start of the occurrence where the items go to).

7.4.10.2 Interaction Declaration

An interaction is declared as a behavior (see 7.4.7), using the keyword **`interaction`**. If no owned subclassification is explicitly given for the interaction, then it is implicitly given default subclassifications to *both* the behavior `Performance` from the `Performances` library model (see 9.2.6) and the association `BinaryLink` or the association `Link` from the `Links` library model (see 9.2.3), depending on whether it is a binary interaction or not.

As a kind of behavior, if the interaction has owned subclassifications whose superclasses are behaviors, then the rules related to their parameters are the same as for any subclassifier behavior (see 7.4.7). As a kind of association, the body of an interaction must declare at least two association ends. If the interaction has owned subclassifications whose superclassifiers are associations, the rules related to their association ends are the same as for any association that is a subclassifier (see 7.4.5).

```
interaction Authorization {
    end feature client[*] : Computer;
    end feature server[*] : Computer;
    composite step login;
    composite step authorize;
    composite succession login then authorize;
}
```

7.4.10.3 Item Flow Declaration

An item flow declaration is syntactically similar to a binary connector declaration (see 7.4.6), using the keyword **`flow`**, or **`succession flow`** for a succession item flow. If no owned subsetting or owned redefinition is explicitly given, then the item flow is implicitly given a default subsetting to the item flow `transfers` from the `Transfers` model library (see 9.2.7), or to the succession item flow `transfersBefore`, if a succession item flow is being declared. If an item flow has owned specializations (including all feature typings, 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 are the same as for the redefinition of the parameters of general behavior or step by a specializing step (see 7.4.7.3).

Unlike a regular binary connector declaration, though, an item flow declaration does not directly specify the related features for the item flow. Instead, the declaration gives the *source output feature* for the transfer after the keyword **from** and the *target input Feature* for the transfer after the keyword **to**. The related features are then determined as the owning features of the features given in the item flow declaration. It is these related features that are constrained to have a common context with the item flow (see 7.4.6), not the features actually given in the declaration.

```
struct Vehicle {
    composite feature fuelTank[1] {
        out feature fuelOut[1] : Fuel;
    }
    composite feature engine {
        in feature fuelIn[1] : Fuel;
    }
    // The item flow actually connects the fuelTank to the engine.
    // The transfer moves Fuel from fuelOut to fuelIn.
    flow fuelFlow from fuelTank::fuelOut to engine::fuelIn;
}
```

The source output and target input features of an item flow can also be specified using feature chains (see 7.3.4.6). In this case, the related features are determined as the features identified by the chains, excluding the last feature. This is particularly useful when the desired related features are inherited features.

```
struct Vehicle {
    composite feature fuelTank[1] {
        out feature fuelOut[1] : Fuel;
    }
    composite feature engine[1] {
        in feature fuelIn[1] : Fuel;
    }
}

feature vehicle : Vehicle {
    // The item flow actually connects the inherited fuelTank
    // feature to the inherited engine feature.
    flow fuelFlow from fuelTank.fuelOut to engine.fuelIn;
}
```

An item flow 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 item flow have the declared type. In the absence of an item declaration, any values may flow across the item flow, consistent with the types of the source output and target input featuees.

```
flow of flowingFuel : Fuel from fuelTank.fuelOut to engine.fuelIn;
```

If no feature declaration or item declaration details are included in an item flow declaration, then the keyword **from** may also be omitted.

```
flow fuelTank.fuelOut to engine.fuelIn;
```

Item flows 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[1] : Image; }
    composite step shoot : Shoot { in image[1] : Image; }
    // The use of a succession item flow means that focus must complete before
    // the image is transferred, after which shoot can begin.
    succession flow focus.image to shoot.image;
}
```

7.4.11 Feature Values

Metamodel references:

- *Concrete syntax,* 8.2.5.10
- *Abstract syntax,* 8.3.4.10
- *Semantics,* 8.4.4.11

A *feature value* is a membership relationship (see 7.2.5) between an owning feature and a *value expression,* whose result provides values for the feature. The feature value relationship is specified as either *bound* or *initial,* and as either *fixed* or a *default.* A feature can have at most one feature value relationship.

A fixed, bound feature value relationship is declared using the symbol = followed by a representation of the value expression using the concrete syntax described in 7.4.9. This notation is appended to the declaration of the owning feature of the feature value.

```
feature monthsInYear : Natural = 12;
struct TestRecord {
    feature scores[1..*] : Integer;
    derived feature averageScore[1] : Rational = sum(scores)/size(scores);
}
```

Features that have a feature value relationship of this form implicitly have a nested binding connector (see 7.4.6) between the feature and the result of the value expression, with the binding connector having the same featuring types as the declared feature (i.e., `TestRecord`, in the example above).

Note. The semantics of binding mean that such a feature value asserts that a feature is *equivalent* to the result of the value expression. To highlight this, a feature with such a feature value 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 feature value – see also 7.3.4.2).

A fixed, initial feature value relationship is declared as above but using the symbol `:=` instead of =.

```
feature count[1] : Natural := 0;
```

In this case, the feature also has an implicit nested binding connector, but the featuring types of the binding connector are the *starting snapshots* of the featuring types of the declared feature. That is, the result of the value expression gives the initial values of the declared feature but, unlike in the case of a bound value, these initial values may subsequently change.

A default feature value relationship is declared similarly to the above, but with the keyword **`default`** preceding the symbol = or `:=`, depending on whether it is bound or initial. However, for a default, bound feature value, the symbol = may be elided.

```
struct Vehicle {
    feature mass[1] : Real default 1500.0;
    feature engine[1] : Engine default := standardEngine;
}
struct TestWithCutoff :> TestRecord {
```

```
        feature cutoff[1] : Rational default = 0.75 * averageScore;
    }
```

For a default feature value relationship, no binding connector is added to the feature declaration, but the default will apply when an instance of the featuring type is constructed, if no other explicit values are given for the feature.

A feature value relationship can be included with the following kinds of feature declaration:

- Feature (see 7.3.4.2)
- Step (see 7.4.7.3)
- Expression (see 7.4.8.3)
- Boolean expression and invariant (see 7.4.8.5)

```
behavior ProvidePower {
    in cmd[1] : Command;
    out wheelTorque[1] : Torque;

    composite step generate : GenerateTorque {
        in cmd = ProvidePower::cmd;
        out generatedTorque;
    }
    composite step apply : ApplyTorque {
        in generatedTorque = generate.generatedTorque;
        out appliedTorque = ProvidePower::wheelTorque;
    }
}
```

7.4.12 Multiplicities

Metamodel references:

- *Concrete syntax,* 8.2.5.11
- *Abstract syntax,* 8.3.4.11
- *Semantics,* 8.4.4.12

Multiplicity is defined in the Core layer as a feature for specifying cardinalities (number of instances) of a type by enumerating all numbers the cardinality might be (see 7.3.2.2). The Kernel layer provides a specific way to do this by specifying a *range* of cardinalities. A multiplicity range has *lower bound* and *upper bound* expressions that are evaluated to determine the lowest and highest cardinalities, with both expression evaluating to natural numbers (that is, of type `Natural` from the `ScalarValues` library model, see 9.3.2). An upper bound value of `*` (infinity) means that the cardinality includes all numbers greater than or equal to the lower bound value.

A multiplicity range is written in the form `[`*`lowerBound`*`..`*`upperBound`*`]`, where each of *`lowerBound`* and *`upperBound`* is either a literal expression or a feature reference expression represented in the notation described in 7.4.9. Literal expressions can be used to specify a multiplicity range with fixed lower and/or upper bounds. If the result of the *`lowerBound`* expression is `*`, then the meaning of the multiplicity range is not defined.

A multiplicity range can also be written without the lower bound (or `..`). In this case, the result of the single 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.

Multiplicity ranges can be used in the declaration of types, particularly features (see 7.3.4.2).

```
struct Automobile {
    feature n : Positive[1];
    composite feature wheels : Wheel[n]; // Equivalent to [n..n] for n < *
```

```
    feature driveWheels[2..n] subsets wheels;
}
feature autoCollection : Automobile[*]; // Equivalent to [0..*]
```

It is also possible to declared a multiplicity feature using the keyword **multiplicity**, optionally followed by a short name and/or name, and including either a multiplicity range or a subsetting of another multiplicity. A multiplicity declaration is a kind of feature declaration, and it can optionally include a body as in a generic feature declaration (see 7.3.4.2).

```
multiplicity zeroOrMore [0..*];
multiplicity m subsets zeroOrMore;
```

If a multiplicity feature is declared in the body of a type, then then this becomes be the multiplicity of the type. A type can have at most one multiplicity, whether this is given in the declaration or the body of the type.

```
feature driveWheels subsets wheels {
    multiplicity [2..n];
}
feature autoCollection {
    multiplicity subsets zeroOrMore;
}
```

7.4.13 Metadata

Metamodel references:

- *Concrete syntax,* 8.2.5.12
- *Abstract syntax,* 8.3.4.12
- *Semantics,* 8.4.4.13

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). In general, metadata is specified in annotating elements (including comments and textual representations) attached to annotated elements (see 7.2.4). A *metadata feature* is a kind of annotating element that allows for the definition of structured metadata with modeler-specified features. This may be used, for example, to add tool-specific information to a model that can be relevant to the function of various kinds of tooling that may use or process a model, or domain-specific information relevant to a certain project or organization.

A metadata feature is syntactically a feature (see 7.3.4) that is typed by a single *metaclass*, which is a kind of structure (see 7.4.4), with implicit multiplicity `1..1`. If the metaclass has no features, then the metadata feature simply acts as a user-defined syntactic tag on the annotated element. If the metaclass has features, then the metadata feature must have nested features that redefine each of the features of its type, binding them to the results of model-level evaluable expressions (see 7.4.9), which provide the values of the specified attributive metadata for the annotated element.

A metaclass is declared like a structure (see 7.4.4), but using the keyword **metaclass**. If no owned subclassification is explicitly given for the metaclass, then it is implicitly given a default subclassification to the metaclass `Metaobject` from the `Metaobjects` library model (see 9.2.16).

```
metaclass SecurityRelated;

metaclass ApprovalAnnotation {
    feature approved[1] : Boolean;
    feature approver[1] : String;
}
```

A metadata feature is declared using the keyword **metadata** (or the symbol @), optionally followed by a short name and/or name, followed by the keyword **typed by** (or the symbol :) and the qualified name of exactly one metaclass. If no short name or name is given, then the keyword **typed by** (or the symbol :) may also be omitted. One or more annotated elements are then identified for the metadata feature after the keyword **about**, indicating that the metadata feature has annotation relationships to each of the identified elements (see 7.2.4).

```
metadata securityDesignAnnotation : SecurityRelated about SecurityDesign;
```

Any owned feature of a metadata feature must be a redefinition of a feature of the typing metaclass, with a feature value binding it to the result of a model-level evaluable expressions (see 7.4.9). The owned features of a metadata feature must always have the same names as the names of the typing metaclass, so the shorthand prefix redefines notation (see 7.3.4.5) 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 metadata feature.

```
metadata ApprovalAnnotation about Design {
    approved = true;
    approver = "John Smith";
}
```

If the metadata feature is an owned member of a namespace (see 7.2.5), then the explicit identification of annotated elements (following the **about** keyword) can be omitted, in which case the annotated element is implicitly the containing namespace (see 7.2.4).

```
class Design {
    // This metadata feature is implicitly about the class Design.
    @ApprovalAnnotation {
        approved = true;
        approver = "John Smith";
    }
}
```

If a metadata feature has one or more concrete features that directly or indirectly subset `Metaobject::annotatedElement`, then, for each annotated element of the metadata feature, there must be at least one such feature for which the metaclass of the annotated element conforms to all the types of the feature (which must all be specializations of the reflective metaclass `KerML::Element`, see 9.2.17).

```
metaclass Command {
    // A metadata feature 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 metadata feature is a direct or indirect specialization of Metaobjects::SemanticMetadata (see 9.2.16.2.3), then the annotated elements must all be types and the feature SemanticMetadata::baseType must be bound to a value of type KerML::Type (see 9.2.17). Each type annotated by such semantic metadata has an implicit specialization added to a type determined from the baseType value as follows:

- If the annotated type is neither a classifier nor a feature, then the annotated type implicitly specializes the baseType.
- If the annotated type is a classifier and the baseType is a classifier, then annotated classifier implicitly subclassifies the baseType.
- If the annotated type is a classifier and the baseType is a feature, then the annotated classifier implicitly subclassifies 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 meta-cast operator **meta** (see 7.4.9.2) 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 meta KerML::Feature" has
    // type KerML::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.
    redefines baseType = userActions meta KerML::Feature;
}

// Save implicitly subclassifies UserAction (which is the
// type of userActions).
behavior Save {
    @Command;
}

// previousAction implicitly subsets userActions.
step previousAction[1] {
    @Command;
}
```

User-Defined Keywords

A *user-defined keyword* is a (possibly qualified) metaclass name or short name preceded by the symbol #. The user-defined keyword is placed immediately before the language-defined (reserved) keyword for the declaration and specifies a metadata feature annotation of the declared element. Note that this notation can only be used for metadata features that do not have nested features. If the named metaclass is a kind of *SemanticMetadata*, then the implicit specialization rules given above for semantic metadata apply.

```
// It is often convenient to use a lower-case initial name or
// short name for semantic metadata intended to be used as a keyword.
metaclass <command> CommandMetadata :> SemanticMetadata {
    redefines baseType = userActions meta KerML::Feature;
```

```
}

#command behavior Save;
#command step previousAction[1];
```

It is also possible to include more than one user defined-keyword in a declaration.

```
#SecurityRelated #command def Save;
```

7.4.14 Packages

Metamodel references:

- *Concrete syntax,* 8.2.5.13
- *Abstract syntax,* 8.3.4.13
- *Semantics,* 8.4.4.14

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). A package is notated like a generic namespace (see 7.2.5.2), but using the keyword **package** instead of **namespace**.

```
package AddressBooks {
    datatype Entry {
        feature name[1]: String;
        feature address[1]: String;
    }
    struct AddressBook {
        composite feature entries[*]: Entry;
    }
}
```

A package may also have one or more *filter conditions* for selecting a subset of its imported memberships. A filter condition is a Boolean-valued, model-level evaluable expression (see 7.4.9) that must evaluate to true for any imported member of the package. These are notated using the keyword **filter** followed by the filter condition expression.

```
package Annotations {
    metaclass ApprovalAnnotation {
        feature approved[1] : Boolean;
        feature approver[1] : String;
        feature level[1] : 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 and
       Annotations::ApprovalAnnotation::approved and
       Annotations::ApprovalAnnotation::level > 1;
}
```

A filter condition can operate on metadata on elements (see 7.4.13), such as checking for a metadata feature of a particular type or accessing the values of the features of a metadata feature. For the purposes of filter condition expressions, every element is also considered to have an implicit metadata feature that is typed by a metaclass from the reflective library model of the KerML abstract syntax (see 9.2.17). This enables filter conditions to test for the abstract syntax metaclass of an element and to access the values of abstract syntax meta-attributes.

Note that a filter condition in a package will filter *all* imports of that Package. That is why full qualification is used for `Annotations::ApprovalAnnotation` in the example above, since imported elements of the `Annotations` package would be filtered out by the very filter condition in which the elements are intended to be used. This may be avoided by combining one or more filter conditions with a specific import, using the filtered import notation described in 7.2.5.4).

```
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 and approved and level > 1];
}
```

The `KerML` library package contains a complete model of the KerML abstract syntax represented in KerML itself. When a filter condition is evaluated on an element, abstract syntax metadata for the element can be tested as if the element had an implicit metadata feature 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 and
                           Structure::ownedFeature != null and
                           @ApprovalAnnotation and
                           ApprovalAnnotation::approved];
}
```

In general, a *library package* is a package that is expected to be commonly available and reused across many user models. A package can be explicitly identified as a library package using the keyword **`library`**. This allows tooling to identify any element contained directly in a library package as being a *library element* from that specific library package.

```
library package AddressBooks {
    ...
}
```

The *standard library packages* in the Kernel Model Libraries (see Clause 9) are further identified using the keyword **`standard`**. However, only library packages from the Kernel Model Libraries, or from other recognized standard model libraries, should be identified as standard library packages.

8 Metamodel

8.1 Metamodel Overview

This clause presents the normative specification of the *metamodel* for KerML, which includes the KerML concrete syntax, abstract syntax and semantics (though the complete semantics depends on the *model library* specified in Clause 9).

1. *Concrete syntax* specifies how the language appears to modelers. Modelers construct and review models using a textual notation that conforms to the concrete syntax specification (see 8.2).
2. *Abstract syntax* specifies linguistic terms and relations between them (as opposed to library model terms), which may be expressed in the concrete syntax (see 8.3). The abstract syntax omits aspects of the concrete syntax, such as delimiters and formatting, that are do not affect what modelers are trying to expression. A concrete syntax representation of a model can be *parsed* into an abstract syntax representation, or an abstract syntax representation can be *serialized* into the concrete syntax notation. The mapping between the concrete and abstract syntax is given as part of the *grammar* specification for the concrete syntax (see 8.2.1 on the conventions for this).
3. *Semantics* specifies the interpretation of models as representations of or specifications for modeled systems (see 8.4). The semantics for a *core* subset of the abstract syntax are specified using mathematical logic. Semantics for the rest of KerML are specified by mapping complicated abstract syntax constructs into equivalent models using the core subset, and, in particular, introducing *implied* relationships to required elements from the KerML model library (see 8.4.1 on this approach).

As described in 6.1, KerML is divided into Root, Core and Kernel Layers, which cut across each of the above facets. The subclauses on Concrete Syntax (8.2) and Abstract Syntax (8.3) are each further subdivided into subclauses on the three layers, and then, within each layer, into subclauses following the package structure of the abstract syntax. Subclause 8.4 on Semantics only covers the Core and Kernel Layers, because Root Layer constructs do not have model-level semantics.

Throughout this clause, the names of elements from the KerML abstract syntax model appear in a "`code`" font. Further:

1. Names of metaclasses 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. 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, when used as English common nouns, e.g., "an `ownedRelatedElement`", "multiple `featuringTypes`", 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."

Similar stylistic conventions apply to text about KerML models, except that an "*`italic code`*" front is used.

1. Convention 1 above applies to KerML `Types` (e.g., *`Performance`*), using "type" (or a more specialized term) instead of "metaclass" (e.g., "the *`Performance`* behavior").
2. Convention 2 above applies to KerML `Features` (e.g, *`performances`*), using "feature" (or a more specialized term) instead of "metaproperty" (e.g., "the *`performances`* step").

8.2 Concrete Syntax

8.2.1 Concrete Syntax Overview

The concrete syntax for KerML is a textual notation that can be used to express or construct an abstract syntax representation of a model. The *lexical structure* of the KerML textual notation defines how the string of characters in a text is divided into a set of *lexical elements*. Such lexical elements can be categorized as *whitespace*, *notes,* or *tokens.* Only tokens are significant for the mapping of the notation to the abstract syntax. The *syntactic structure* of the KerML textual notation defines how lexical tokens are grouped and mapped to an abstract syntax representation of a model.

Both the lexical and syntactic structures are specified as *grammars* consisting of productions for lexical elements or non-terminal syntactic elements (see Table 1). The body of a production is specified using an Extended Backus Naur Form (EBNF) notation (see Table 2). The syntactic grammar includes further notations to describe how the concrete syntax maps to the abstract syntax element being synthesized (see Table 3).

Subclause 8.2.2 presents the lexical grammar for KerML. Subclauses 8.2.3, 8.2.4, and 8.2.5 then each present the portion of the syntactic grammar for KerML covering the Root, Core and Kernel Layers of KerML (see 6.1). Each of these subclauses is further divided into subclauses corresponding to each of the packages from the abstract syntax model (see 8.3). The starting production for the syntactic grammar is `RootNamespace` (see 8.2.3.4.1).

Table 1. Grammar Production Definitions

`LEXICAL_ELEMENT = ...`	Define a production for the `LEXICAL_ELEMENT`.
`NonterminalElement : AbstractSyntaxElement = ...`	Define a production for the `NonterminalElement` that synthesizes the `AbstractSyntaxElement`. If the `NonterminalElement` has the same name as the `AbstractSyntaxElement`, then ": `AbstractSyntaxElement`" may be omitted.

Table 2. EBNF Notation Conventions

Lexical element	`LEXICAL_ELEMENT`
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

Property assignment	`p = Element`	Assign the result of parsing the concrete syntax `Element` to abstract syntax property `p`.

List property construction	`p += Element`	Add the result of parsing the concrete syntax `Element` to the abstract syntax list property `p`.
Boolean property assignment	`p ?= Element`	If the concrete syntax `Element` is parsed, then set the abstract Boolean property `p` to true.
Non-parsing assignment	`{ p = value }` `{ p += value }`	Assign (or add) the given `value` to the abstract syntax property `p`, without parsing any input. The `value` may be a literal or a reference to another abstract syntax property. The symbol "`this`" refers to the element being synthesized.
Name resolution	`[QualifiedName]`	Parse a `QualifiedName`, then resolve that name to an `Element` reference (see 8.2.3.5) for use as a value in an assignment as above.

8.2.2 Lexical Structure

8.2.2.1 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
```

Notes

1. Notation text is 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.

2. Any characters in text line that are not a part of the line terminator are referred to as *line text*.

3. 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 8.2.2.2).

8.2.2.2 Notes and Comments

```
SINGLE_LINE_NOTE =
    '//' LINE_TEXT

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

REGULAR_COMMENT =
    '/*' COMMENT_TEXT '*/'

COMMENT_TEXT =
    ( COMMENT_LINE_TEXT | LINE_TERMINATOR )*

COMMENT_LINE_TEXT =
    LINE_TEXT excluding the sequence '*/'
```

8.2.2.3 Names

```
NAME =
    BASIC_NAME | UNRESTRICTED_NAME

BASIC_NAME =
    BASIC_INITIAL_CHARACTER BASIC_NAME_CHARACTER*

UNRESTRICTED_NAME =
    single_quote ( NAME_CHARACTER | ESCAPE_SEQUENCE )* single_quote
    (see Note 1)

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 Note 2
```

Notes

1. The `single_quote` character is `'`. The name represented by an `UNRESTRICTED_NAME` shall consist of the characters *within* the single quotes, with escape characters resolved as described below. The surrounding single quote characters are *not* part of the represented name.
2. An `ESCAPE_SEQUENCE` is a sequence of two text characters starting with a backslash that 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 also 8.2.2.1). Table 4 shows the meaning of the allowed escape sequences. The `ESCAPE_SEQUENCES` in an `UNRESTRICTED_NAME` shall be replaced by the characters specified as their meanings in the actual represented name.

Table 4. Escape Sequences

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

8.2.2.4 Numeric Values

```
DECIMAL_VALUE =
    DECIMAL_DIGIT+

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

Notes

1. A `DECIMAL_VALUE` may specify a natural literal, or it may be part of the specification of a real literal (see 8.2.5.8.4). Note that a `DECIMAL_VALUE` does not include a sign, because negating a literal is an operator in the KerML `Expression` syntax.
2. An `EXPONENTIAL_VALUE` may be used in the specification of a real literal (see 8.2.5.8.4). 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.

8.2.2.5 String Value

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

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

Notes

1. `ESCAPE_SEQUENCE` is specified in 8.2.2.3.

8.2.2.6 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 dependency derived differences disjoining disjoint doc else`
`end expr false feature featured featuring filter first flow for from function`

```
hastype if intersects implies import in inout interaction inv inverse inverting
istype language member metaclass metadata multiplicity namespace nonunique not
null of or ordered out package portion predicate private protected public
readonly redefines redefinition references rep return specialization specializes
step struct subclassifier subset subsets subtype succession then to true type
typed typing unions xor
```

Tooling for the KerML textual notation should generally highlight keywords relative to other text, for example by using boldface and/or distinctive coloring. However, while keywords are shown in boldface in this specification, the specification does not require any specific highlighting (or any highlighting at all), and KerML textual notation documents are expected to be interchanged as plain text (see also Clause 10 on Model Interchange).

8.2.2.7 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 8.2.5.8) 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'
REFERENCES   = '::>' | 'references'
REDEFINES    = ':>>' | 'redefines'
CONJUGATES   = '~'   | 'conjugates'
```

8.2.3 Root Concrete Syntax

8.2.3.1 Elements and Relationships Concrete Syntax

```
Identification : Element =
    ( '<' declaredShortName = NAME '>' )?
    ( declaredName = NAME )?

RelationshipBody : Relationship =
    ';' | '{' RelationshipOwnedElement* '}'

RelationshipOwnedElement : Relationship =
      ownedRelatedElement += OwnedRelatedElement
    | ownedRelationship += OwnedAnnotation

OwnedRelatedElement : Element =
    NonFeatureElement | FeatureElement
```

8.2.3.2 Dependencies Concrete Syntax

```
Dependency =
    ( ownedRelationship += PrefixMetadataAnnotation )*
    'dependency' ( Identification? 'from' )?
    client += [QualifiedName] ( ',' client += [QualifiedName] )* 'to'
    supplier += [QualifiedName] ( ',' supplier += [QualifiedName] )*
    RelationshipBody
```

Notes

1. `PrefixMetadataAnnotation` is defined in the Kernel layer (see 8.2.5.12).

8.2.3.3 Annotations Concrete Syntax

8.2.3.3.1 Annotations

```
Annotation =
    annotatedElement = [QualifiedName]

OwnedAnnotation : Annotation =
    annotatingElement = AnnotatingElement
    { ownedRelatedElement += annotatingElement }

AnnotatingElement =
      Comment
    | Documentation
    | TextualRepresentation
    | MetadataFeature
```

Notes

1. `MetadataFeature` is defined in the Kernel layer (see 8.2.5.12).

8.2.3.3.2 Comments and Documentation

```
Comment =
    'comment' Identification
    ( 'about' annotation += Annotation
      { ownedRelationship += annotation }
      ( ',' annotation += Annotation
        { ownedRelationship += annotation } )*
    )?
    body = REGULAR_COMMENT

Documentation =
    'doc' Identification
    body = REGULAR_COMMENT
```

Notes

1. The text of a lexical `REGULAR_COMMENT` or `PREFIX_COMMENT` shall be processed as follows before it is included as the body of a `Comment` or `Documentation`:

    1. Remove the initial `/*` and final `*/` characters.

    2. Remove any white space immediately after the initial `/*`, up to and including the first line terminator (if any).

3. On each subsequent line of the text:

    1. Strip initial white space other than line terminators.

    2. Then, if the first remaining character is "*", remove it.

    3. Then, if the first remaining character is now a space, remove it.

2. 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.

8.2.3.3.3 Textual Representation

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

Notes

1. The lexical text of a `REGULAR_COMMENT` shall be processed as specified in 8.2.3.3.2 for `Comments` before being included as the body of a `TextualRepresentation`.
2. See also 8.3.2.3.6 on the standard `language` names recognized for a `TextualRepresentation`.

8.2.3.4 Namespaces Concrete Syntax

8.2.3.4.1 Namespaces

```
RootNamespace : Namespace =
    NamespaceBodyElement*
(See Note 1)

Namespace =
    ( ownedRelationship += PrefixMetadataMember )*
    NamespaceDeclaration NamespaceBody
(See Note 2)

NamespaceDeclaration : Namespace =
    'namespace' Identification

NamespaceBody : Namespace =
    ';' | '{' NamespaceBodyElement* '}'

NamespaceBodyElement : Namespace =
      ownedRelationship += NamespaceMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import

MemberPrefix : Membership =
    ( visibility = VisibilityIndicator )?

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

NamespaceMember : OwningMembership =
```

```
      NonFeatureMember
    | NamespaceFeatureMember

NonFeatureMember : OwningMembership =
    MemberPrefix
    ownedRelatedElement += MemberElement

NamespaceFeatureMember : OwningMembership =
    MemberPrefix
    ownedRelatedElement += FeatureElement

AliasMember : Membership =
    MemberPrefix
    'alias' ( '<' memberShortName = NAME '>' )?
    ( memberName = NAME )?
    'for' memberElement = [QualifiedName]
    RelationshipBody

QualifiedName =
   ( NAME '::' )* NAME
(See Note 3)
```

Notes

1. A *root* `Namespace` is a `Namespace` that has no `owningNamespace` (see 8.3.2.4). Every `Element` other than a root `Namespace` must be contained, directly or indirectly, within some root `Namespace`. Therefore, every valid KerML concrete syntax text can be parsed starting from the `RootNamespace` production.
2. `PrefixMetadataMember` is defined in the Kernel layer (see 8.2.5.12).
3. 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. The segment names of the qualified name other than the last identify a sequence of nested `Namespaces` that provide the context for resolving the final segment name (see 8.2.3.5). The notation `[QualifiedName]` is used in concrete syntax grammar productions to indicate the result of resolving text parsed as a `QualifiedName` (see also 8.2.1).

8.2.3.4.2 Imports

```
Import =
    ( visibility = VisibilityIndicator )?
    'import' ( isImportAll ?= 'all' )?
    ImportDeclaration RelationshipBody

ImportDeclaration : Import
    MembershipImport | NamespaceImport

MembershipImport =
    importedMembership = [QualifiedName]
    ( '::' isRecursive ?= '**' )?
(see Note 1)

NamespaceImport =
      importedNamespace = [QualifiedName] '::' '*'
      ( '::' isRecursive ?= '**' )?
```

```
    | importedNamespace = FilterPackage
      { ownedRelatedElement += importedNamespace }

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

FilterPackageMember : ElementFilterMembership =
    '[' ownedRelatedElement += OwnedExpression ']'
    { visibility = 'private' }
```

Notes

1. The `importedMembership` of a `MembershipImport` is the single case in which the `Element` required from the resolution `[QualifiedName]` is the actual `Membership` identified by the `QualifedName`, *not* the `memberElement` of that `Membership` (see 8.2.3.5).

8.2.3.4.3 Namespace Elements

```
MemberElement : Element =
    AnnotatingElement | NonFeatureElement

NonFeatureElement : Element =
      Dependency
    | Namespace
    | Type
    | Classifier
    | DataType
    | Class
    | Structure
    | Metaclass
    | Association
    | AssociationStructure
    | Interaction
    | Behavior
    | Function
    | Predicate
    | Multiplicity
    | Package
    | LibraryPackage
    | Specialization
    | Conjugation
    | Subclassification
    | Disjoining
    | FeatureInverting
    | FeatureTyping
    | Subsetting
    | Redefinition
    | TypeFeaturing

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

```
| ItemFlow
| SuccessionItemFlow
```

8.2.3.5 Name Resolution

8.2.3.5.1 Name Resolution Overview

A qualified name consists of a sequence of one or more *segment names* (see 8.2.3.4.1). Each segment names is a *simple name,* that is, it is a lexical `NAME` token (see 8.2.2.3). The *qualification part* of a qualified name with more than one segment name is itself a qualified name, consisting of all the segment names of the original qualified name except for the last. For example the qualified name `A::B::C` consists of the segment names `A`, `B` and `C`, and its qualification part is `A::B`.

Name resolution is a process for determining the `Element` that is identified by a qualified name. The result of the process is actually a `Membership` relationship identified by the qualified name. However, in all cases but one, the required `Element` to be inserted into the abstract syntax is the `memberElement` of that `Membership`, in which case the metaclass of the `memberElement` must conform to the expected metaclass in the context of the name resolution. The one exception is the resolution of the qualified name for the `importedMembership` of a `MembershipImport` (see 8.2.3.4.2), in which case the required `Element` is the identified `Membership` itself.

The *basic* name resolution process consists of the following two steps. The terms "local Namespace", "visible resolution" and "full resolution" used below are defined in 8.2.3.5.2, 8.2.3.5.3, and 8.2.3.5.4.

1. If the qualified name has only one segment name, then the resolution of the qualified name is the *full resolution* of that segment name relative to the *local* `Namespace` for the qualified name – unless the local `Namespace` is a root `Namespace`, in which case the global `Namespace` is used instead.
2. Otherwise, resolve the qualification part of the qualified name relative to the local `Namespace` of the original qualified name. This must resolve to a `Namespace`, and the resolution of the original qualified name is then the *visible resolution* of its last segment name relative to this `Namespace`.

If either of the above steps fails, or if the resulting `Element` does not have the proper type for its context, then the qualified name has no resolution, and the parsing of the text containing it fails with a *name resolution error.*

Note. The invoking the `Namespace::resolve`, as defined in the abstract syntax (see 8.3.2.4.5), carries out the above basic resolution process with the target `Namespace` considered as the local `Namespace` for the given qualified name.

The basic name resolution process is used directly to resolve a qualified name in all cases *except* when the qualified name specifies the `redefinedFeature` of a `Redefinition` with an `owningFeature` that has an `owningType`. In this case, the basic name resolution processes is repeated with the general Type of each `ownedSpecialization` of the `owningType` considered in turn as the local `Namespace`, until a resolution is found. If no resolution is found for any of these, then the overall resolution fails.

Note. When implementing the name resolution process as specified here, some additional points need to be considered.

- The descriptions given in 8.2.3.5.2, 8.2.3.5.3, and 8.2.3.5.4 presume that the derived `membership`, `importedMembership` and (for a `Type`) `inheritedMembership` properties of a `Namespace` have been fully computed, including `memberships` resulting from implied `Relationships` (see 8.4.2). However, when parsing a complete KerML concrete syntax text, the values of these properties may themselves be based on other `Relationships` (e.g., alias `Memberships`, `Imports` and `Specializations`) whose target references are given by qualified names that must be resolved. Name resolution must therefore proceed incrementally during a parse, avoiding infinite loops caused by attempting to resolve again names that are already pending resolution. Note, however, that it *is* possible to at least locally resolve a name to a Membership in a Namespace without immediately resolving the memberElement of that Membership.

- Circularity is allowed for Imports and Specializations. Therefore, when traversing the graph of these Relationships, an implementation must avoid re-processing a Namespace that has already been visited.

8.2.3.5.2 Local and Global Namespaces

Every Namespace other than a root Namespace (see 8.2.3.4.1) is nested in a containing Namespace called its owningNamespace (see 8.3.2.4).

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 root Namespaces from the KerML Model Libraries (see Clause 9). If a tool imports a *model interchange project* (see 10.3), then the available Namespaces shall also include all the root Namespaces from any used project of the imported project. A conforming tool can also provide means for making additional Namespaces available to a root Namespace, such as by creating a new root Namespace or adding an additional used project.

A qualified name is always used to identify an Element that is a target Element of some *context* Relationship. The *local* Namespace for resolving the qualified name is then determined depending on the kind of context Relationship, as given in the following.

Import (see 8.3.2.4.2)

- The local Namespace is the importOwningNamespace.

Membership (see 8.3.2.4.3)

- If the membershipOwningNamespace is a FeatureReferenceExpression (see 8.3.4.8.4), then the local Namespace is the *non-invocation* Namespace for the membershipOwningNamespace, which is defined to be the nearest containing Namespace that is none of the following:
    - FeatureReferenceExpression
    - InvocationExpression
    - ownedFeature of an InvocationExpression
- If the membershipOwningNamespace is a FeatureChainExpression see 8.3.4.8.3, then the local Namespace is the result parameter of the argument Expression of the FeatureChainExpression.
- Otherwise, the local Namespace is the membershipOwningNamespace.

Specialization (see 8.3.3.1.8)

- If the Specialization is a ReferenceSubsetting (see 8.3.3.3.9), and its referencingFeature is an end Feature whose owningType is a Connector, then the local Namespace is the owningNamespace of the Connector.
- If the Specialization is a FeatureTyping (see 8.3.3.3.6), and its owningFeature is an InvocationExpression, then the local Namespace is the non-invocation Namespace for the owningFeature (determined as for a FeatureReferenceExpression under Membership above).
- Otherwise, if the owningType is not null, then the local Namespace is the owningNamespace of the owningType.
- Otherwise, the local Namespace is the owningNamespace of the Specialization.

Conjugation (see 8.3.3.1.2)

- If the owningType is not null, the local Namespace is the owningNamespace of the owningType.
- Otherwise, the local Namespace is the owningNamespace of the Conjugation.

FeatureChaining (see 8.3.3.3.4)

- If the FeatureChaining is the first ownedFeatureChaining of its featureChained, then the local Namespace is determined as if the owningRelationship of the featureChained (which will be a Membership, Subsetting or Conjugation) was the context Relationship (see above).
- Otherwise, the local Namespace is the chainingFeature of the previous FeatureChaining in the ownedFeatureChaining list.

8.2.3.5.3 Local and Visible Resolution

A Namespace defines a mapping from names to its memberships, know as the *local resolution* of those names. Each membership of a Namespace is the local resolution for its memberShortName and memberName (if non-null). Note that this includes owned, imported and (if the Namespace is a Type) inherited Memberships.

Note. If the Namespace is well formed, then there can be at most one Membership that is the local resolution of any given name.

The *visible resolution* of a name is similar to its local resolution, but the memberships considered are restricted to those that are *visible* outside the Namespace. The *visible* Memberships of a Namespace shall comprise the following:

- All ownedMemberships of the Namespace with visibility = public.
- All importedMemberships of the Namespace that are derived from Import Relationships with visibility = public.
- If the Namespace is a Type, then all inheritedMemberships of the Type with visibility = public.

8.2.3.5.4 Full Resolution

The *full resolution* of a simple name relative to a Namespace considers Memberships not only in that Namespace, but also in directly or indirectly containing Namespaces, all the way out to the global Namespace. Full resolution relative to a Namespace *other* than the global Namespace proceeds as follows:

1. If the name has a local resolution relative to a Namespace (see 8.2.3.5.3), then that is also its full resolution relative to that Namespace.
2. Otherwise:
   - If the Namespace is *not* a root Namespace, then the full resolution of the name relative to the original Namespace is determined as its full resolution relative to the owningNamespace of the original Namespace.
   - If the Namespace *is* a root Namespace, then the full resolution of the name resolution relative to the original Namespace is its resolution in the global Namespace.

The resolution of a simple name in the global Namespace is the the Membership in the global Namespace (as defined in 8.2.3.5.2) whose (non-null) shortMemberName or memberName is equal to the simple name.

Note. It is possible that there will be more than one Membership in the global Namespace that resolves 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.

8.2.4 Core Concrete Syntax

8.2.4.1 Types Concrete Syntax

8.2.4.1.1 Types

```
Type =
    TypePrefix 'type'
    TypeDeclaration TypeBody

TypePrefix : Type =
    ( isAbstract ?= 'abstract' )?
    ( ownedRelationship += PrefixMetadataMember )*

TypeDeclaration : Type =
    ( isSufficient ?= 'all' )? Identification
    ( ownedRelationship += OwnedMultiplicity )?
    ( SpecializationPart | ConjugationPart )+
    TypeRelationshipPart*

SpecializationPart : Type =
    SPECIALIZES ownedRelationship += OwnedSpecialization
    ( ',' ownedRelationship += OwnedSpecialization )*

ConjugationPart : Type =
    CONJUGATES ownedRelationship += OwnedConjugation

TypeRelationshipPart : Type =
      DisjoiningPart
    | UnioningPart
    | IntersectingPart
    | DifferencingPart

DisjoiningPart : Type =
    'disjoint' 'from' ownedRelationship += OwnedDisjoining
    ( ',' ownedRelationship += OwnedDisjoining )*

UnioningPart : Type =
    'unions' ownedRelationship += Unioning
    ( ',' ownedRelationship += Unioning )*

IntersectingPart : Type =
    'intersects' ownedRelationship += Intersecting
    ( ',' ownedRelationship += Intersecting )*

DifferencingPart : Type =
    'differences' ownedRelationship += Differencing
    ( ',' ownedRelationship += Differencing )*

TypeBody : Type =
    ';' | '{' TypeBodyElement* '}'

TypeBodyElement : Type =
      ownedRelationship += NonFeatureMember
    | ownedRelationship += FeatureMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import
```

8.2.4.1.2 Specialization

```
Specialization =
    ( 'specialization' Identification )?
```

```
    'subtype' SpecificType
    SPECIALIZES GeneralType
    RelationshipBody

OwnedSpecialization : Specialization =
    GeneralType

SpecificType : Specialization :
      specific = [QualifiedName]
    | specific += OwnedFeatureChain
      { ownedRelatedElement += specific }

GeneralType : Specialization =
      general = [QualifiedName]
    | general += OwnedFeatureChain
      { ownedRelatedElement += general }
```

8.2.4.1.3 Conjugation

```
Conjugation =
    ( 'conjugation' Identification )?
    'conjugate'
    ( conjugatedType = [QualifiedName]
    | conjugatedType = FeatureChain
      { ownedRelatedElement += conjugatedType }
    )
    CONJUGATES
    ( originalType = [QualifiedName]
    | originalType = FeatureChain
      { ownedRelatedElement += originalType }
    )
    RelationshipBody

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

8.2.4.1.4 Disjoining

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

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

8.2.4.1.5 Unioning, Intersecting and Differencing

```
Unioning =
      unioningType = [QualifiedName]
    | ownedRelatedElement += OwnedFeatureChain

Intersecting =
      intersectingType = [QualifiedName]
    | ownedRelatedElement += OwnedFeatureChain

Differencing =
      differencingType = [QualifiedName]
    | ownedRelatedElement += OwnedFeatureChain
```

8.2.4.1.6 Feature Membership

```
FeatureMember : OwningMembership =
      TypeFeatureMember
    | OwnedFeatureMember

TypeFeatureMember : OwningMembership =
    MemberPrefix 'member' ownedRelatedElement += FeatureElement

OwnedFeatureMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += FeatureElement
```

8.2.4.2 Classifiers Concrete Syntax

8.2.4.2.1 Classifiers

```
Classifier =
    TypePrefix 'classifier'
    ClassifierDeclaration TypeBody

ClassifierDeclaration : Classifier =
    ( isSufficient ?= 'all' )? Identification
    ( ownedRelationship += OwnedMultiplicity )?
    ( SuperclassingPart | ConjugationPart )?
    RelationshipPart*

SuperclassingPart : Classifier =
      SPECIALIZES ownedRelationship += OwnedSubclassification
      ( ',' ownedRelationship += OwnedSubclassification )*
```

8.2.4.2.2 Subclassification

```
Subclassification =
    ( 'specialization' Identification )?
    'subclassifier' subclassifier = [QualifiedName]
    SPECIALIZES superclassifier = [QualifiedName]
    RelationshipBody

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

8.2.4.3 Features Concrete Syntax

8.2.4.3.1 Features

```
Feature =
    FeaturePrefix
    ( 'feature'? FeatureDeclaration
    | 'feature'
    | ownedRelationship += PrefixMetadataMember
    )
    ValuePart? TypeBody
(See Note 1)

FeaturePrefix : Feature =
    ( direction = FeatureDirection )?
    ( isAbstract ?= 'abstract' )?
    ( isComposite ?= 'composite' | isPortion ?= 'portion' )?
    ( isReadOnly ?= 'readonly' )?
    ( isDerived ?= 'derived' )?
    ( isEnd ?= 'end' )?
    ( ownedRelationship += PrefixMetadataMember )*
(See Note 1)

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

FeatureDeclaration : Feature =
    ( isSufficient ?= 'all' )?
    ( FeatureIdentification
      ( FeatureSpecializationPart | ConjugationPart )?
    | FeatureSpecializationPart
    | FeatureConjugationPart
    )
    FeatureRelationshipPart*

FeatureIdentification : Feature =
      '<' shortName = NAME '>' ( name = NAME )?
    | name = NAME

FeatureRelationshipPart : Feature =
      TypeRelationshipPart
    | ChainingPart
    | InvertingPart
    | TypeFeaturingPart

ChainingPart : Feature =
    'chains'
    ( ownedRelationship += OwnedFeatureChaining
    | FeatureChain )

InvertingPart : Feature =
    'inverse' 'of' ownedRelationship += OwnedFeatureInverting

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

FeatureSpecializationPart : Feature =
      FeatureSpecialization+ MultiplicityPart? FeatureSpecialization*
    | MultiplicityPart FeatureSpecialization*
```

```
MultiplicityPart : Feature =
      ownedRelationship += OwnedMultiplicity
    | ( ownedRelationship += OwnedMultiplicity )?
      ( isOrdered ?= 'ordered' ( {isUnique = false} 'nonunique' )?
      | {isUnique = false} 'nonunique' ( isOrdered ?= 'ordered' )? )

FeatureSpecialization : Feature =
    Typings | Subsettings | References | Redefinitions

Typings : Feature =
      TypedBy ( ',' ownedRelationship += OwnedFeatureTyping )*

TypedBy : Feature =
    TYPED_BY ownedRelationship += OwnedFeatureTyping

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

Subsets : Feature =
    SUBSETS ownedRelationship += OwnedSubsetting

References : Feature =
    REFERENCES ownedRelationship += OwnedReferenceSubsetting

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

Redefines : Feature =
    REDEFINES ownedRelationship += OwnedRedefinition
```

Notes

1. `PrefixMetadataMember` is defined in the Kernel layer (see 8.3.4.12).

8.2.4.3.2 Feature Typing

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

OwnedFeatureTyping : FeatureTyping =
    GeneralType
```

8.2.4.3.3 Subsetting

```
Subsetting =
    ( 'specialization' Identification )?
    'subset' SpecificType
    SUBSETS GeneralType
    RelationshipBody

OwnedSubsetting : Subsetting =
    GeneralType

OwnedReferenceSubsetting : ReferenceSubsetting =
    GeneralType
```

8.2.4.3.4 Redefinition

```
Redefinition =
    ( 'specialization' Identification )?
    'redefinition' SpecificType
    REDEFINES GeneralType
    RelationshipBody

OwnedRedefinition : Redefinition =
    GeneralType
```

8.2.4.3.5 Feature Chaining

```
OwnedFeatureChain : Feature =
    FeatureChain

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

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

8.2.4.3.6 Feature Inverting

```
FeatureInverting =
    ( 'inverting' Identification? )?
    'inverse'
    ( featureInverted = [QualifiedName]
    | featureInverted = OwnedFeatureChain
      { ownedRelatedElement += featureInverted }
    )
    'of'
    ( invertingFeature = [QualifiedName]
    | ownedRelatedElement += OwnedFeatureChain
      { ownedRelatedElement += invertingFeature }
    )
    RelationshipBody

OwnedFeatureInverting : FeatureInverting =
      invertingFeature = [QualifiedName]
    | invertingFeature = OwnedFeatureChain
      { ownedRelatedElement += invertingFeature }
```

8.2.4.3.7 Type Featuring

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

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

8.2.5 Kernel Concrete Syntax

8.2.5.1 Data Types Concrete Syntax

```
DataType =
    TypePrefix 'datatype'
    ClassifierDeclaration TypeBody
```

8.2.5.2 Classes Concrete Syntax

```
Class =
    TypePrefix 'class'
    ClassifierDeclaration TypeBody
```

8.2.5.3 Structures Concrete Syntax

```
Structure =
    TypePrefix 'struct'
    ClassifierDeclaration TypeBody
```

8.2.5.4 Associations Concrete Syntax

```
Association =
    TypePrefix 'assoc'
    ClassifierDeclaration TypeBody

AssociationStructure =
    TypePrefix 'assoc' 'struct'
    ClassifierDeclaration TypeBody
```

8.2.5.5 Connectors Concrete Syntax

8.2.5.5.1 Connectors

```
Connector =
    FeaturePrefix 'connector'
    ConnectorDeclaration TypeBody

ConnectorDeclaration : Connector =
    BinaryConnectorDeclaration | NaryConnectorDeclaration

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

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

ConnectorEndMember : EndFeatureMembership =
    ownedRelatedElement += ConnectorEnd

ConnectorEnd : Feature =
    ( declaredName = NAME REFERENCES )?
    ownedRelationship += OwnedReferenceSubsetting
    ( ownedRelationship += OwnedMultiplicity )?
```

8.2.5.5.2 Binding Connectors

```
BindingConnector =
    FeaturePrefix 'binding'
    BindingConnectorDeclaration TypeBody

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

8.2.5.5.3 Successions

```
Succession =
    FeaturePrefix 'succession'
    SuccessionDeclaration TypeBody

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

8.2.5.6 Behaviors Concrete Syntax

8.2.5.6.1 Behaviors

```
Behavior =
    TypePrefix 'behavior'
    ClassifierDeclaration TypeBody
```

8.2.5.6.2 Steps

```
Step =
    FeaturePrefix
    'step' FeatureDeclaration ValuePart?
    TypeBody
```

8.2.5.7 Functions Concrete Syntax

8.2.5.7.1 Functions

```
Function =
    TypePrefix 'function'
    ClassifierDeclaration FunctionBody

FunctionBody : Type =
    ';' | '{' FunctionBodyPart '}

FunctionBodyPart : Type =
     ( TypeBodyElement
     | ownedRelationship += ReturnFeatureMember
     )*
   ( ownedRelationship += ResultExpressionMember )?
```

```
ReturnFeatureMember : ReturnParameterMembership =
    MemberPrefix 'return'
    ownedRelatedElement += FeatureElement

ResultExpressionMember : ResultExpressionMembership =
    MemberPrefix
    ownedRelatedElement += OwnedExpression
```

8.2.5.7.2 Expressions

```
Expression =
    FeaturePrefix
    'expr' FeatureDeclaration ValuePart?
    FunctionBody
```

8.2.5.7.3 Predicates

```
Predicate =
    TypePrefix 'predicate'
    ClassifierDeclaration FunctionBody
```

8.2.5.7.4 Boolean Expressions and Invariants

```
BooleanExpression =
    FeaturePrefix
    'bool' FeatureDeclaration ValuePart?
    FunctionBody

Invariant =
    FeaturePrefix
    'inv' ( 'true' | isNegated ?= 'false' )?
    FeatureDeclaration ValuePart?
    FunctionBody
```

8.2.5.8 Expressions Concrete Syntax

8.2.5.8.1 Operator Expressions

```
OwnedExpressionReferenceMember : FeatureMembership =
    ownedRelationship += OwnedExpressionReference

OwnedExpressionReference : FeatureReferenceExpression =
    ownedRelationship += OwnedExpressionMember

OwnedExpressionMember : FeatureMembership =
    ownedFeatureMember = OwnedExpression

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

ConditionalExpression : OperatorExpression =
      operator = 'if'
      ownedRelationship += ArgumentMember '?'
      ownedRelationship += ArgumentExpressionMember 'else'
```

```
    ownedRelationship += ArgumentExpressionMember

ConditionalBinaryOperatorExpression : OperatorExpression =
    ownedRelationship += ArgumentMember
    operator = ConditionalBinaryOperator
    ownedRelationship += ArgumentExpressionMember

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

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

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

UnaryOperatorExpression : OperatorExpression =
    operator = UnaryOperator
    ownedRelationship += ArgumentMember

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

ClassificationExpression : OperatorExpression =
    ( ownedRelationship += ArgumentMember )?
    ( operator = ClassificationTestOperator
      ownedRelationship += TypeReferenceMember
    | operator = CastOperator
      ownedRelationship += TypeResultMember
    )

ClassificationTestOperator =
    'istype' | 'hastype' | '@'

CastOperator =
    'as'

MetaclassificationExpression : OperatorExpression =
    ownedRelationship += MetadataArgumentMember
    ( operator = MetaClassificationTestOperator
      ownedRelationship += TypeReferenceMember
    | operator = MetaCastOperator
      owendRelationsip += TypeResultMember
    )

ArgumentMember : ParameterMembership =
    ownedMemberParameter = Argument

Argument : Feature =
    ownedRelationship += ArgumentValue

ArgumentValue : FeatureValue =
    value = OwnedExpression
```

```
ArgumentExpressionMember : FeatureMembership =
    ownedRelatedElement += ArgumentExpression

ArgumentExpression : Feature =
    ownedRelationship += ArgumentExpressionValue

ArgumentExpressionValue : FeatureValue =
    value = OwnedExpressionReference

MetadataArgumentMember : ParameterMembership =
    ownedRelatedElement += MetadataArgument

MetadataArgument : Feature =
    ownedRelationship += MetadataValue

MetadataValue : FeatureValue =
    value = MetadataReference

MetadataReference : MetadataAccessExpression =
    referencedElement = [QualifiedName]

MetaclassificationOperator =
   '@@' | 'meta'

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

TypeReferenceMember : FeatureMembership =
    ownedMemberFeature = TypeReference

TypeResultMember : ResultParameterMembership =
    ownedMemberFeature = TypeReference

TypeReference : Feature =
    ownedRelationship += ReferenceTyping

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

Notes

1. `OperatorExpressions` provide a shorthand notation for `InvocationExpressions` that invoke a library `Function` represented as an `operator` *symbol.* Table 5 shows the mapping from `operator` symbols to the `Functions` they represent from the Kernel Model Library (see Clause 9). An `OperatorExpression` contains subexpressions called its *operands* that generally correspond to the `argument Expressions` of the `OperatorExpression`, except in the case of `operators` representing *control* `Functions`, in which case the evaluation of certain operands is as determined by the `Function` (see 8.4.4.9 for details).
2. Though not directly expressed in the syntactic productions given above, in any `OperatorExpression` containing nested `OperatorExpressions`, the nested `OperatorExpressions` shall be implicitly grouped according to the *precedence* of the `operators` involved, as given in Table 6. `OperatorExpressions` with higher precedence `operators` shall be grouped more tightly than those with lower precedence `operators`.
3. The unary `operator` symbol ~ maps to the library `Function` *`DataFunctions::'~'`*, as shown in Table 5. This abstract `Function` may be given a concrete definition in a domain-specific `Function` library, but

no default definition is provided in the Kernel Functions Library. If no domain-specific definition is available, a tool should give a warning if this `operator` is used.

Table 5. Operator Mapping

Operator	Library Function	Description	Model-Level Evaluable?
`all`	`BaseFunctions::'all'`	Type extent	No
`istype`	`BaseFunctions::'istype'`	All argument values are directly or indirectly instances of a type	Yes
`hastype`	`BaseFunctions::'hastype'`	All argument values are directly instances of a type	Yes
`@`	`BaseFunctions::'@'`	Any argument value is directly or indirectly an instance of a type	Yes
`@@`	`BaseFunctions::'@@'`	Any argument value is directly or indirectly an instance of a metaclass	Yes
`as`	`BaseFunctions::as`	Select instances of type (cast)	Yes
`meta`	`BaseFunctions::meta`	Select instances of a metaclass (metacast)	Yes
`==`	`BaseFunctions::'=='`	Equality	Yes
`!=`	`BaseFunctions::'!='`	Inequality	Yes
`===`	`BaseFunctions::'==='`	Same (equality for data values, same lives for occurrences)	Yes
`!==`	`BaseFunctions::'!=='`	Not same	Yes
`xor`	`DataFunctions::'xor'`	Logical "exclusive or"	Yes
`not`	`DataFunctions::'not'`	Logical "not"	Yes
`~`	`DataFunctions::'~'`	Undefined	No
`|`	`DataFunctions::'|'`	Logical "inclusive or"	Yes
`&`	`DataFunctions::'&'`	Logical "and"	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

Operator	Library Function	Description	Model-Level Evaluable?
%	DataFunctions::'%'	Remainder	Yes
^ **	DataFunctions::'^'	Exponentiation	Yes
..	DataFunctions::'..'	Range construction	Yes
??	ControlFunctions::'??'	Null coalescing	Yes
if	ControlFunctions::'if'	Conditional test (ternary)	Yes
or	ControlFunctions::'or'	Conditional "or"	Yes
and	ControlFunctions::'and'	Conditional "and"	Yes
implies	ControlFunctions::'implies'	Conditional "implication"	Yes

Table 6. Operator Precedence (highest to lowest)

Unary
all
+ - ~ **not**
Binary
^ **
* / %
+ -
..
< > <= >=
istype hastype @ @@ **as meta**
== != === !==
& **and**
xor
| **or**
implies
??

Ternary
if

8.2.5.8.2 Primary Expressions

```
PrimaryExpression : Expression =
      FeatureChainExpression
    | NonFeatureChainPrimaryExpression

PrimaryExpressionMember : FeatureMembership =
    ownedMemberFeature = PrimaryExpression

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

NonFeatureChainPrimaryExpressionMember : FeatureMembership =
    ownedMemberFeature = NonFeatureChainPrimaryExpression

BracketExpression : OperatorExpression =
    ownedRelationship += PrimaryExpressionMember
    operator = '['
    ownedRelationship += SequenceExpressionListMember ']'

IndexExpression : OperatorExpression =
    ownedRelationship += PrimaryExpressionMember
    operator = '#'
    '(' ownedRelationsip += SequenceExpressionListMember ')'

SequenceExpression : Expression =
    '(' SequenceExpressionList ')'

SequenceExpressionList : Expression =
    OwnedExpression ','? | SequenceOperatorExpression

SequenceOperatorExpression : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = ','
    ownedRelationship += SequenceExpressionListMember

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
```

Notes

1. 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 7. Note also that, for a `CollectionExpression` or `SelectExpression`, the abstract syntax constrains the `operator` to be *`collect`* and *`select`*, respectively, separately from the `.` and `.?` symbols used in their concrete syntax notation (see 8.3.4.8.2 and 8.3.4.8.15).
2. The grammar allows a bracket syntax `[...]` that parses to an invocation of the library `Function` *`BaseFunctions::'['`*, as shown in Table 7. This notation is available for use with domain-specific library models that given a concrete definition to the abstract base `'['` `Function`, but no default definition is provided in the Kernel Functions Library. If no domain-specific definition is available, a tool should give a warning if this operator is used.

Table 7. Primary Expression Operator Mapping

Operator	Library Function	Description	Model-level Evaluable?
`[`	`BaseFunctions::'['`	Undefined	No
`#`	`BaseFunctions::'#'`	Indexing	Yes
`,`	`BaseFunctions::','`	Sequence construction	Yes
`.`	`ControlFunctions::'.'`	Feature chaining	Yes
`collect`	`ControlFunctions::collect`	Sequence collection	Yes

Operator	Library Function	Description	Model-level Evaluable?
`select`	`ControlFunctions::select`	Sequence selection	Yes

8.2.5.8.3 Base Expressions

```
BaseExpression : Expression =
      NullExpression
    | LiteralExpression
    | FeatureReferenceExpression
    | MetadataAccessExpression
    | InvocationExpression
    | BodyExpression

NullExpression : NullExpression =
    'null' | '(' ')'

FeatureReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FeatureReferenceMember

FeatureReferenceMember : Membership =
    memberElement = FeatureReference

FeatureReference : Feature =
    [QualifiedName]

MetadataAccessExpression =
    referenceElement = [QualifiedName] '.' 'metadata'

InvocationExpression : InvocationExpression =
    ownedRelationship += ( OwnedFeatureTyping | OwnedSubsetting )
    ArgumentList
(See Note 1)

ArgumentList : InvocationExpression =
    '(' ( PositionalArgumentList | NamedArgumentList )? ')'

PositionalArgumentList : InvocationExpression =
    e.ownedRelationship += ArgumentMember
    ( ',' e.ownedRelationship += ArgumentMember )*

NamedArgumentList : InvocationExpression =
    ownedRelationship += NamedArgumentMember
    ( ',' ownedRelationship += NamedArgumentMember )*

NamedArgumentMember : FeatureMembership =
    ownedMemberFeature = NamedArgument

NamedArgument : Feature =
    ownedRelationship += ParameterRedefinition '='
    ownedRelationship += ArgumentValue

ParameterRedefinition : Redefinition =
    redefinedFeature = [QualifiedName]

BodyExpression : FeatureReferenceExpression =
    ownedRelationship += ExpressionBodyMember

ExpressionBodyMember : FeatureMembership =
```

```
    ownedMemberFeature = ExpressionBody

ExpressionBody : Expression =
    '{' FunctionBodyPart '}'
```

Notes

1. The first `ownedRelationship` of an `InvocationExpression` should be parsed as a `FeatureTyping` if the target `Type` is a `Classifier` and as a `Subsetting` if the target `Type` is a `Feature`.

8.2.5.8.4 Literal Expressions

```
LiteralExpression =
      LiteralBoolean
    | LiteralString
    | LiteralInteger
    | LiteralReal
    | LiteralInfinity

LiteralBoolean =
    value = BooleanValue

BooleanValue : Boolean =
    'true' | 'false'

LiteralString =
    value = STRING_VALUE

LiteralInteger =
    value = DECIMAL_VALUE

LiteralReal =
    value = RealValue

RealValue : Real =
      DECIMAL_VALUE? '.' ( DECIMAL_VALUE | EXPONENTIAL_VALUE )
    | EXPONENTIAL_VALUE

LiteralInfinity =
    '*'
```

8.2.5.9 Interactions Concrete Syntax

8.2.5.9.1 Interactions

```
Interaction =
    TypePrefix 'interaction'
    ClassifierDeclaration TypeBody
```

8.2.5.9.2 Item Flows

```
ItemFlow =
    FeaturePrefix 'flow'
    ItemFlowDeclaration TypeBody

SuccessionItemFlow =
    FeaturePrefix 'succession' 'flow'
    ItemFlowDeclaration TypeBody
```

```
ItemFlowDeclaration : ItemFlow =
    ( FeatureDeclaration ValuePart?
      ( 'of' ownedRelationship += ItemFeatureMember )?
      ( 'from' ownedRelationship += ItemFlowEndMember
        'to' ownedRelationship += ItemFlowEndMember )?
    | ( isSufficient ?= 'all' )?
      ownedRelationship += ItemFlowEndMember 'to'
      ownedRelationship += ItemFlowEndMember

ItemFeatureMember : FeatureMembership =
    ownedRelatedElement = ItemFeature

ItemFeature : Feature =
      Identification ItemFeatureSpecializationPart ValuePart?
    | ( ownedRelationship += OwnedFeatureTyping
      ( ownedRelationship += OwnedMultiplicity )?
    | ownedRelationship += OwnedMultiplicity
      ( ownedRelationship += OwnedFeatureTyping )?

ItemFeatureSpecializationPart : Feature =
      FeatureSpecialization+ MultiplicityPart?
      FeatureSpecialization*
    | MultiplicityPart FeatureSpecialization+

ItemFlowEndMember : EndFeatureMembership =
    ownedRelatedElement += ItemFlowEnd

ItemFlowEnd : ItemFlowEnd =
    ( ownedRelationship += OwnedReferenceSubsetting '.' )?
    ownedRelationship += ItemFlowFeatureMember

ItemFlowFeatureMember : FeatureMembership =
    ownedRelatedElement += ItemFlowFeature

ItemFlowFeature : Feature =
    ownedRelationship += ItemFlowRedefinition

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

8.2.5.10 Feature Values Concrete Syntax

```
ValuePart : Feature =
    ownedRelationship += FeatureValue

FeatureValue =
    ( '='
    | isInitial ?= ':='
    | isDefault ?= 'default' ( '=' | isInitial ?= ':=' )?
    )
    ownedRelatedElement += OwnedExpression
```

8.2.5.11 Multiplicities Concrete Syntax

```
Multiplicity =
    MultiplicitySubset | MultiplicityRange

MultiplicitySubset : Multiplicity =
    'multiplicity' Identification Subsets
    TypeBody
```

```
MultiplicityRange =
    'multiplicity' Identification MultiplicityBounds
    TypeBody

OwnedMultiplicity : OwningMembership =
    ownedRelatedElement += OwnedMultiplicityRange

OwnedMultiplicityRange : MultiplicityRange =
    MultiplicityBounds

MultiplicityBounds : MultiplicityRange =
    '[' ( ownedRelationship += MultiplicityExpressionMember '..' )?
          ownedRelationship += MultiplicityExpressionMember ']'

MultiplicityExpressionMember : OwningMembership =
    ownedRelatedElement += ( LiteralExpression | FeatureReferenceExpression )
```

8.2.5.12 Metadata Concrete Syntax

```
Metaclass =
    TypePrefix 'metaclass'
    ClassifierDeclaration TypeBody

PrefixMetadataAnnotation : Annotation =
    '#' ownedRelatedElement += PrefixMetadataFeature

PrefixMetadataMember : OwningMembership =
    '#' ownedRelatedElement += PrefixMetadataFeature

PrefixMetadataFeature : MetadataFeature :
    ownedRelationship += OwnedFeatureTyping

MetadataFeature =
    ( '@' | 'metadata' )
    MetadataFeatureDeclaration
    ( 'about' annotation += Annotation
      { ownedRelationship += annotation }
      ( ',' annotation += Annotation
        { ownedRelationship += annotation } )*
    )?
    MetadataBody

MetadataFeatureDeclaration : MetadataFeature =
    ( Identification ( ':' | 'typed' 'by' ) )?
    ownedRelationship += OwnedFeatureTyping

MetadataBody : Feature =
    ';' | '{' ( ownedRelationship += MetadataBodyElement )* '}'

MetadataBodyElement : Membership =
      NonFeatureMember
    | MetadataBodyFeatureMember
    | AliasMember
    | Import

MetadataBodyFeatureMember : FeatureMembership =
    ownedMemberFeature = MetadataBodyFeature

MetadataBodyFeature : Feature =
```

```
    'feature'? ( ':>>' | 'redefines')? ownedRelationship += OwnedRedefinition
    FeatureSpecializationPart? ValuePart?
    MetadataBody
```

8.2.5.13 Packages Concrete Syntax

```
Package =
    ( ownedRelationship += PrefixMetadataMember )*
    PackageDeclaration PackageBody

LibraryPackage =
    ( isStandard ?= 'standard' ) 'library'
    ( ownedRelationship += PrefixMetadataMember )*
    PackageDeclaration PackageBody

PackageDeclaration : Package =
    'package' Identification

PackageBody : Package =
      ';'
    | '{' ( NamespaceBodyElement
          | ownedRelationship += ElementFilterMember
          )*
      '}'

ElementFilterMember : ElementFilterMembership =
    MemberPrefix
    'filter' condition = OwnedExpression ';'
```

8.3 Abstract Syntax

8.3.1 Abstract Syntax Overview

The KerML abstract syntax is specified as a UML model conforming to the CMOF conformance point of the Meta Object Facility Core Specification [MOF]. As shown in Fig. 1, this model is divided into three top-level packages corresponding to the three layers of KerML (see 8.1). Each top-level package contains nested packages for the modeling areas it addresses. Further, the Core package imports the Root package and the Kernel package imports the Core package, so that the Kernel package contains (as owned or imported members) all abstract syntax elements. Fig. 2 shows the generalization hierarchy for all abstract syntax elements, other than those that represent KerML Relationships, and Fig. 3 shows a similar hierarchy for all abstract syntax elements that represent Relationships.

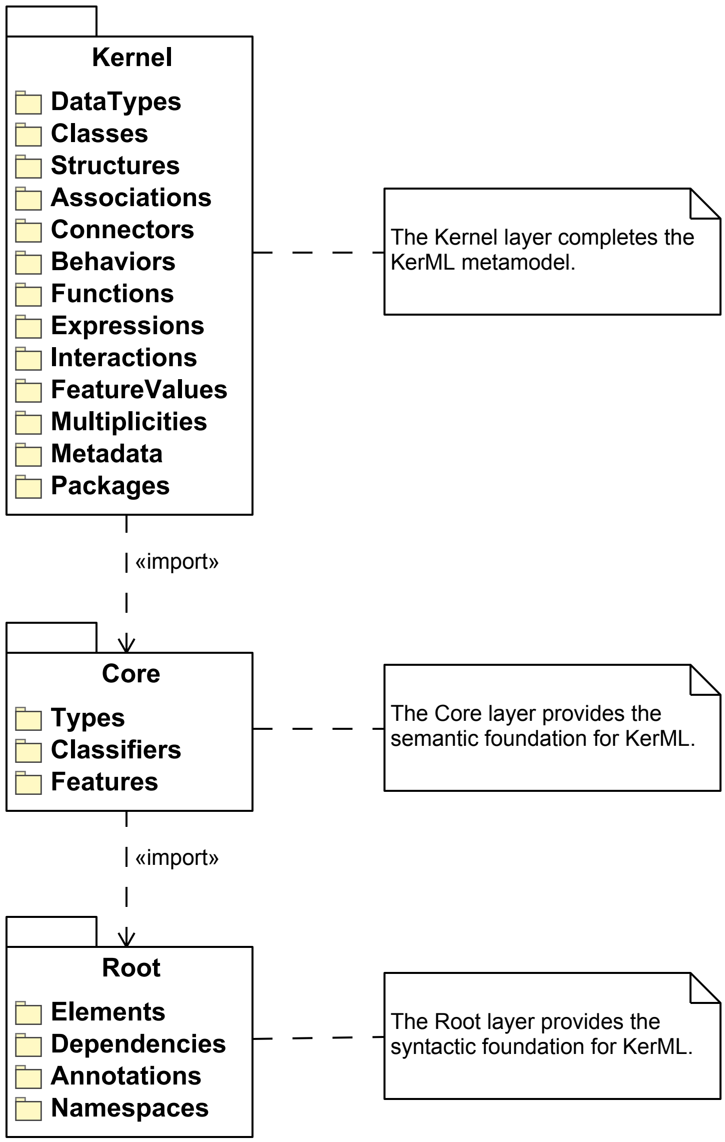

Figure 1. KerML Syntax Layers

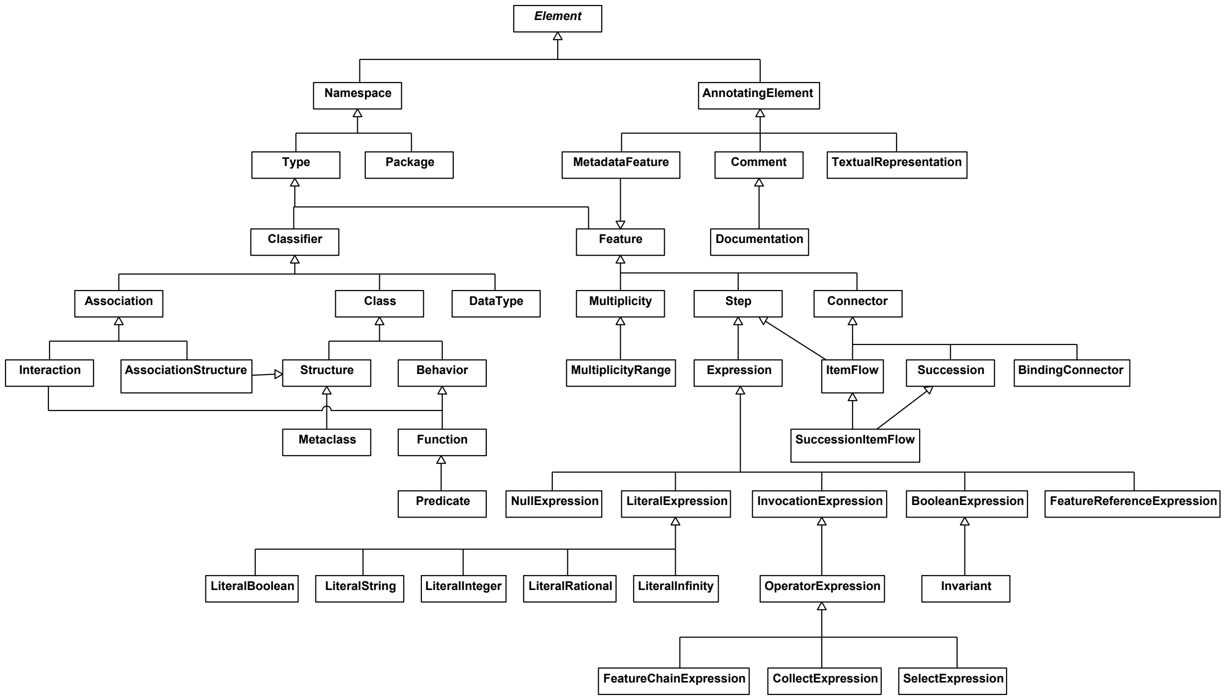

Figure 2. KerML Element Hierarchy

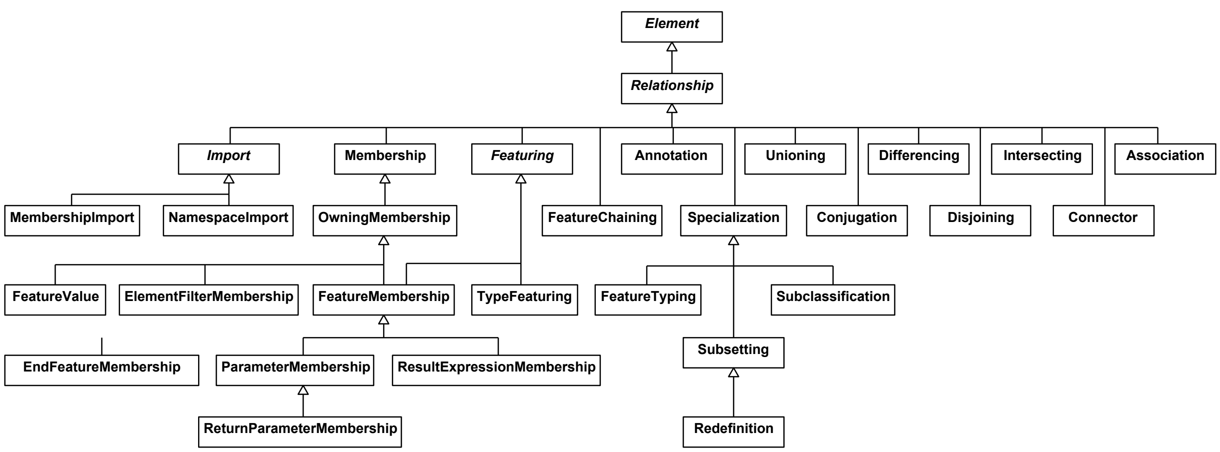

Figure 3. KerML Relationship Hierarchy

The MOF-compliant class model for the abstract syntax defines the basic structural representation for any KerML model. It is also the basis for the textual concrete syntax (see 8.2) and for other forms of serialization used for interchanging models (see Clause 10). In addition to this basic structure, the abstract syntax also includes *constraints* defined on various metaclasses. A conformant tool shall be able to accept any KerML model that conforms to the structural abstract syntax class model, and it may then additionally report on and/or enforce the constraints on a model so represented (as further described below).

The abstract syntax model includes three kinds of constraints:

1. *Derivation constraints.* These constraints specify the how the values of the derived properties of a metaclass are computed from the values of other properties in the abstract syntax model. A tool conformant to the KerML abstract syntax shall always enforce derivation constraints. However, the computed values of derived properties may depend on whether implied relationships are included in the

model or not (see below). A derivation constraint has a name starting with the word `derive`, followed by the name of the metaclass it constrains, followed by the name of the derived property it is for. The OCL specification of such a constraint always has the form of an equality, with the derived property on the left-hand side and the derivation expression on the right-hand side. For example, the derivation constraint for the derived property `Element::ownedElement` is called `deriveElementOwnedElement` and has the OCL specification `ownedElement = ownedRelationship.ownedRelatedElement`.

Note. Derivation constraints are *not* included for derived properties in the following cases:

   - The derived property subsets a property with multiplicity upper bound 1. In this case, if the derived property has a value, it must be the same as that of the subsetted property.
   - The derived property redefines another derived property. In this case, the derivation of the redefined property also applies to the redefining property, though the redefining property will generally place additional constraints on type and/or multiplicity.

2. *Semantic constraints.* These constraints specify relationships that are semantically required in a KerML model (see 8.4.2), particularly relationships with elements in the Kernel Semantic Library (see 9.2). These constraints may be violated by a model as entered by a user or as interchanged. In this case, a tool may satisfy the constraints by introducing *implied relationships* into the model, it may simply report their violation, or it may ignore the violations. Semantic constraints have names that start with the word `check`, followed by the name of the constrained metaclass, followed by a descriptive word or phrase. For example, `checkTypeSpecialization`.
3. *Validation constraints.* These constraints specify additional syntactic conditions that must be satisfied in order to give a model a proper semantic interpretation. They are written presuming that all semantic constraints are satisfied. A *valid* model is a model that satisfies all validation constraints. A tool conformant to the KerML abstract syntax should report violations of validation constraints. A tool conformant to the KerML semantics is only required to operate on valid models. Validation constraints have names that start with the word `validate`, followed by the name of the metaclass, followed by a descriptive word or phrase. For example, `validateConnectorRelatedFeatures`.

8.3.2 Root Abstract Syntax

8.3.2.1 Elements and Relationships Abstract Syntax

8.3.2.1.1 Overview

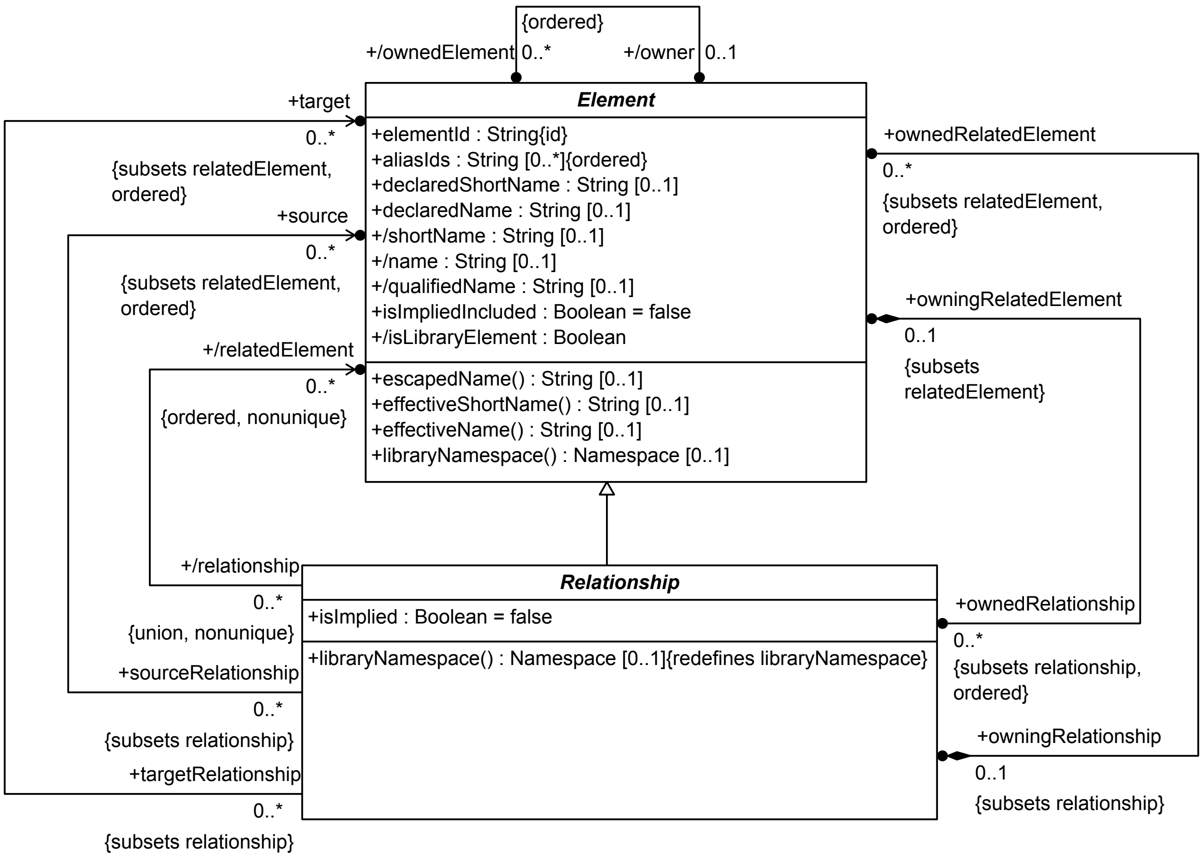

Figure 4. Elements

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.

8.3.2.1.2 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.

declaredName : String [0..1]

The declared name of this `Element`.

declaredShortName : 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.

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

The Documentation owned by this Element.

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.

isImpliedIncluded : Boolean

Whether all necessary implied Relationships have been included in the `ownedRelationships` of this Element. This property may be true, even if there are not actually any `ownedRelationships` with `isImplied = true`, meaning that no such Relationships are actually implied for this Element. However, if it is false, then `ownedRelationships` may *not* contain any implied Relationships. That is, either *all* required implied Relationships must be included, or none of them.

/isLibraryElement : Boolean

Whether this Element is contained in the ownership tree of a library model.

/name : String [0..1]

The name to be used for this `Element` during name resolution within its `owningNamespace`. This is derived using the `effectiveName()` operation. By default, it is the same as the `declaredName`, but this is overridden for certain kinds of `Elements` to compute a `name` even when the `declaredName` is null.

/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`, which is 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]

The short name to be used for this `Element` during name resolution within its `owningNamespace`. This is derived using the `effectiveShortName()` operation. By default, it is the same as the `declaredShortName`, but this is overridden for certain kinds of `Elements` to compute a `shortName` even when the `declaredName` is null.

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

The `TextualRepresentations` that annotate this `Element`.

Operations

effectiveName() : String [0..1]

Return an effective `name` for this `Element`. By default this is the same as its `declaredName`.

```
body: declaredName
```

effectiveShortName() : String [0..1]

Return an effective `shortName` for this `Element`. By default this is the same as its `declaredShortName`.

```
body: declaredShortName
```

escapedName() : String [0..1]

Return `name`, if that is not null, otherwise the `shortName`, if that is not null, otherwise null. If the returned value 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).

libraryNamespace() : Namespace [0..1]

By default, return the library Namespace of the `owningRelationship` of this Element, if it has one.

```
body: if owningRelationship <> null then owningRelationship.libraryNamespace()
else null endif
```

Constraints

deriveElementDocumentation

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

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

deriveElementIsLibraryElement

An `Element` `isLibraryElement` if `libraryNamespace()` is not null.

```
isLibraryElement = libraryNamespace() <>null
```

deriveElementName

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

```
name = effectiveName()
```

deriveElementOwnedAnnotation

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

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

deriveElementOwnedElement

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

```
ownedElement = ownedRelationship.ownedRelatedElement
```

deriveElementOwner

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

```
owner = owningRelationship.owningRelatedElement
```

deriveElementQualifiedName

If this `Element` does not have an `owningNamespace`, then its `qualifiedName` is null. 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-null 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 null or the escaped name is null, in which case the `qualifiedName` of this `Element` is also null.

```
qualifiedName =
    if owningNamespace = null then null
```

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

deriveElementShortName

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

```
shortName = effectiveShortName()
```

deriveElementTextualRepresentation

The `textualRepresentations` of an `Element` are its `ownedElements` that are `TextualRepresentations`.

```
textualRepresentation = ownedElement->selectByKind(TextualRepresentation)
```

deriveOwningNamespace

The `owningNamespace` of an `Element` is the `membershipOwningNamspace` of its `owningMembership` (if any).

```
owningNamespace =
    if owningMembership = null then null
    else owningMembership.membershipOwningNamespace
    endif
```

validateElementIsImpliedIncluded

If an `Element` has any `ownedRelationships` for which `isImplied = true`, then the `Element` must also have `isImpliedIncluded = true`. (Note that an `Element` *can* have `isImplied = true` even if no `ownedRelationships` have `isImplied = true`, indicating the `Element` simply has no implied `Relationships`.

```
ownedRelationship->exists(isImplied) implies isImpliedIncluded
```

8.3.2.1.3 Relationship

Description

A `Relationship` is an `Element` that relates other `Element`. 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 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 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 `Relationship`.

General Classes

Element

Attributes

isImplied : Boolean

Whether this Relationship was generated by tooling to meet semantic rules, rather than being directly created by a modeler.

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 [0..*] {ordered, nonunique}

The Elements that are related by this Relationship, derived as the union of the `source` and `target` Elements of the Relationship.

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

libraryNamespace() : Namespace [0..1]

Return whether this Relationship has either an `owningRelatedElement` or `owningRelationship` that is a library element.

```
body: if owningRelatedElement <> null then owningRelatedElement.libraryNamespace()
else if owningRelationship <> null then owningRelationship.libraryNamespace()
else null endif endif
```

Constraints

deriveRelationshipRelatedElement

The `relatedElements` of a `Relationship` consist of all of its `source Elements` followed by all of its `target Elements`.

```
relatedElement = source->union(target)
```

8.3.2.2 Dependencies Abstract Syntax

8.3.2.2.1 Overview

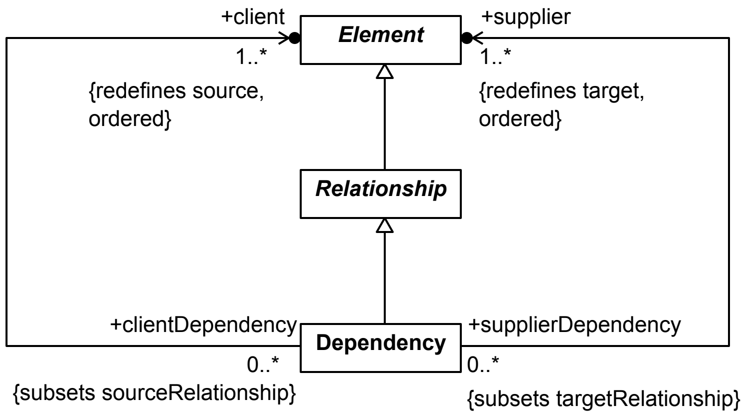

Figure 5. Dependencies

8.3.2.2.2 Dependency

Description

A `Dependency` is a `Relationship` that indicates that one or more `client Elements` require one more `supplier Elements` for their complete specification. In general, this means that a change to one of the `supplier Elements` may necessitate a change to, or re-specification of, the `client Elements`.

Note that a `Dependency` is entirely a model-level `Relationship`, without instance-level semantics.

General Classes

Relationship

Attributes

client : Element [1..*] {redefines source, ordered}

The `Element` or `Elements` dependent on the `supplier Elements`.

supplier : Element [1..*] {redefines target, ordered}

The `Element` or `Elements` on which the `client Elements` depend in some respect.

Operations

None.

Constraints

None.

8.3.2.3 Annotations Abstract Syntax

8.3.2.3.1 Overview

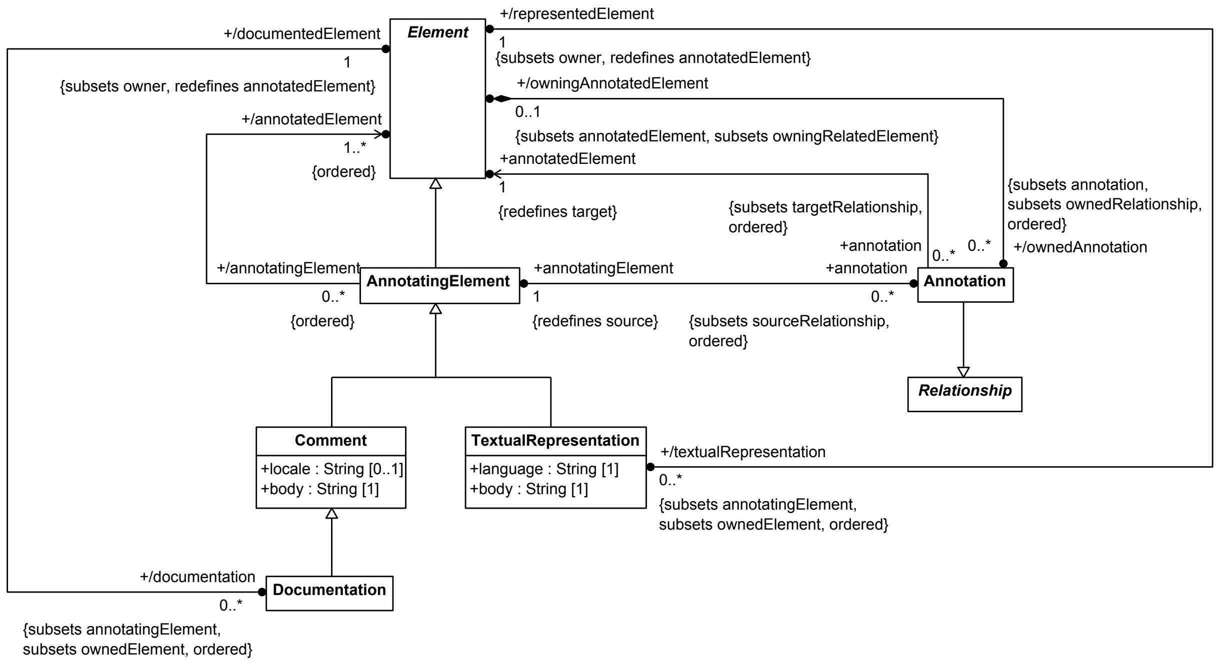

Figure 6. Annotation

The `textualAnnotations` of an `Element` are its `annotatingElements` that are `TextualRepresentations`.

8.3.2.3.2 AnnotatingElement

Description

An `AnnotatingElement` is an `Element` that provides additional description of or metadata on some other `Element`. An `AnnotatingElement` is either attached to its `annotatedElements` by `Annotation Relationships`, or it implicitly annotates its `owningNamespace`.

General Classes

Element

Attributes

/annotatedElement : Element [1..*] {ordered}

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

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

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

Operations

None.

Constraints

deriveAnnotatingElementAnnotatedElement

If an `AnnotatingElement` has `annotations`, then its `annotatedElements` are the `annotatedElements` of all its `annotations`. Otherwise, it's single `annotatedElement` is its `owningNamespace`.

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

8.3.2.3.3 Annotation

Description

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

General Classes

Relationship

Attributes

annotatedElement : Element {redefines target}

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

annotatingElement : AnnotatingElement {redefines source}

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

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

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

Operations

None.

Constraints

None.

8.3.2.3.4 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

None.

Constraints

None.

8.3.2.3.5 Documentation

Description

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

General Classes

Comment

Attributes

/documentedElement : Element {subsets owner, redefines annotatedElement}

The `Element` that is documented by this `Documentation`.

Operations

None.

Constraints

None.

8.3.2.3.6 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 artificial language, in which case the `body` is expected to be a formal, machine-parsable representation.

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. The following `language` names are defined to correspond to the given standard languages:

kerml	Kernel Modeling Language
ocl	Object Constraint Language
alf	Action Language for fUML

Other specifications may define specific language strings, other than those shown above, to be used to indicate the use of languages from those specifications in KerML TextualRepresentation.

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. 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.

An Element with a TextualRepresentation 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.

General Classes

AnnotatingElement

Attributes

body : String

The textual representation of the representedElement in the given language.

language : String

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

/representedElement : Element {subsets owner, redefines annotatedElement}

The Element that is represented by this TextualRepresentation.

Operations

None.

Constraints

None.

8.3.2.4 Namespaces Abstract Syntax

8.3.2.4.1 Overview

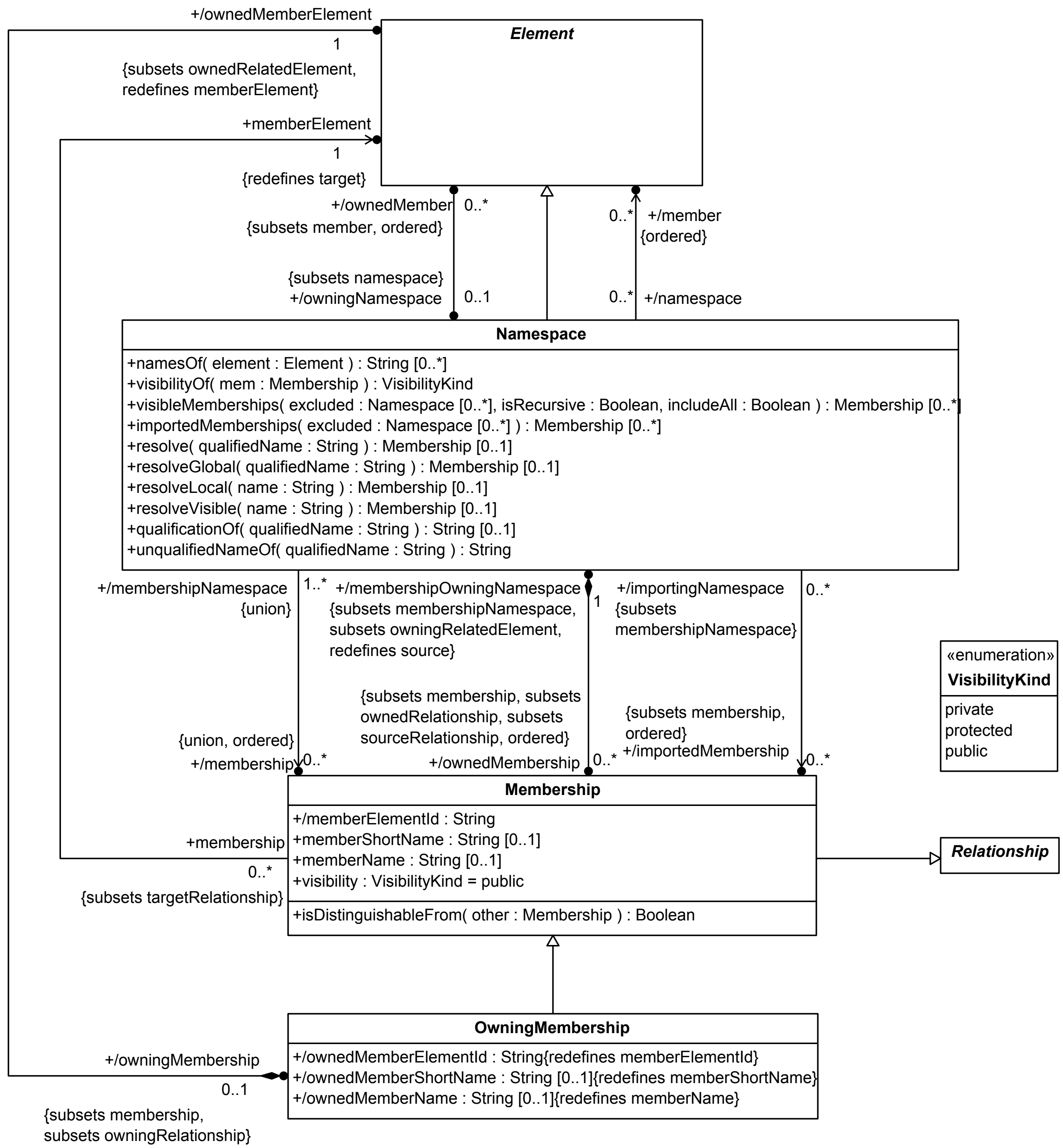

Figure 7. Namespaces

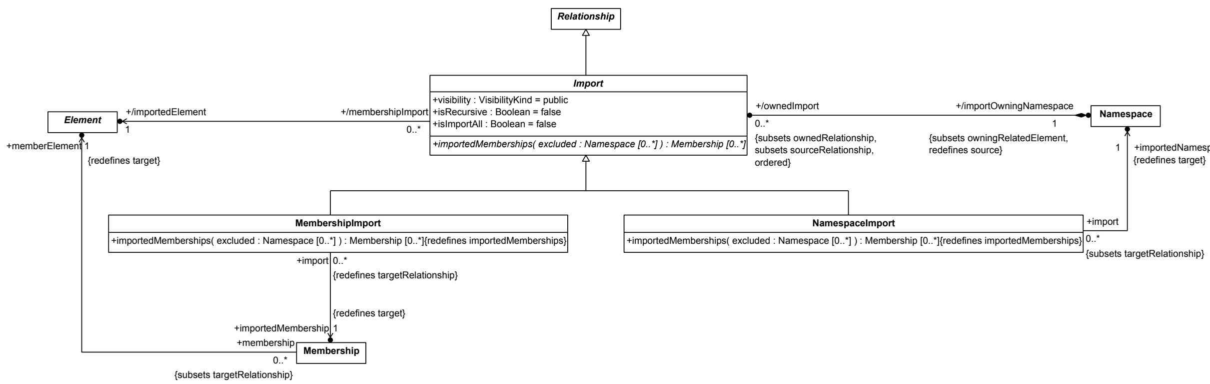

Figure 8. Imports

8.3.2.4.2 Import

Description

An `Import` is an `Relationship` between its `importOwningNamespace` and either a `Membership` (for a `MembershipImport`) or another `Namespace` (for a `NamespaceImport`), which determines a set of `Memberships` that 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 `isRecursive = true`, then visible `Memberships` are also recursively imported from owned sub-`Namespaces`.

General Classes

Relationship

Attributes

/importedElement : Element

The effectively imported `Element` for this Import. For a `MembershipImport`, this is the `memberElement` of the `importedMembership`. For a `NamespaceImport`, it is the `importedNamespace`.

/importOwningNamespace : Namespace {subsets owningRelatedElement, redefines source}

The Namespace into which Memberships 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

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

Returns Memberships that are to become `importedMemberships` of the `importOwningNamespace`. (The `excluded` parameter is used to handle the possibility of circular Import Relationships.)

Constraints

None.

8.3.2.4.3 Membership

Description

A `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

deriveMembershipMemberElementId

The `memberElementId` of a `Membership` is the `elementIf` of its `memberElement`.

```
memberElementId = memberElement.elementId
```

8.3.2.4.4 MembershipImport

Description

A `MembershipImport` is an `Import` that imports its `importedMembership` into the `importOwningNamespace`. If `isRecursive = true` and the `memberElement` of the `importedMembership` is a `Namespace`, then the equivalent of a recursive `NamespaceImport` is also performed on that `Namespace`.

General Classes

Import

Attributes

importedMembership : Membership {redefines target}

The `Membership` to be imported.

Operations

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

Returns at least the `importedMembership`. If `isRecursive = true` and the `memberElement` of the `importedMembership` is a `Namespace`, then `Memberships` are also recursively imported from that `Namespace`.

```
body: if not isRecursive or
   not importedElement.oclIsKindOf(Namespace) or
   excluded->includes(importedElement)
then Sequence{importedMembership}
```

```
else importedElement.oclAsType(Namespace).
        visibleMemberships(excluded, true, importAll)->
        prepend(importedMembership)
endif
```

Constraints

deriveMembershipImportImportedElement

The `importedElement` of a `MembershipImport` is the `memberElement` of its `importedMembership`.

```
importedElement = importedMembership.memberElement
```

8.3.2.4.5 Namespace

Description

A `Namespace` is an `Element` that contains other `Element`, 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 `Namespace`.

A `Namespace` can provide names for its `members` via the `memberNames` and `memberShortNames` specified by the `Memberships` in the `Namespace`. If a `Membership` specifies a `memberName` and/or `memberShortName`, then that those are names of the corresponding `memberElement` relative to the `Namespace`. For an `OwningMembership`, the `owningMemberName` and `owningMemberShortName` are given by the `Element name` and `shortName`. 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 the `ownedImports` of this `Namespace`.

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

The set of all member `Elements` of this `Namespace`, which are 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, which are the ownedMemberElements of the ownedMemberships of the .

/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.importedMemberships(excluded->including(self))
```

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()
```

qualificationOf(qualifiedName : String) : String [0..1]

Return a string with valid KerML syntax representing the qualification part of a given qualifiedName, that is, a qualified name with all the segment names of the given name except the last. If the given qualifiedName has only one segment, then return null.

```
body: No OCL
```

resolve(qualifiedName : String) : Membership [0..1]

Resolve the given qualified name to the named Membership (if any), starting with this Namespace as the local scope. The qualified name string must conform to the concrete syntax of the KerML textual notation. According to the KerML name resolution rules every qualified name will resolve to either a single Membership, or to none.

```
body: let qualification : String = qualificationOf(qualifiedName) in
let name : String = unqualifiedNameOf(qualifiedName) in
if qualification = null then resolveLocal(name)
else
    let namespace : Element = resolve(qualification) in
    if namespace = null or not namespace.oclIsKindOf(Namespace) then null
    else namespace.oclAsType(Namespace).resolveVisible(name) endif
endif
```

resolveGlobal(qualifiedName : String) : Membership [0..1]

Resolve the given qualified name to the named Membership (if any) in the effective global Namespace that is the outermost naming scope. The qualified name string must conform to the concrete syntax of the KerML textual notation.

```
body: No OCL
```

resolveLocal(name : String) : Membership [0..1]

Resolve a simple `name` starting with this `Namespace` as the local scope, and continuing with containing outer scopes as necessary. However, if this `Namespace` is a root `Namespace`, then the resolution is done directly in global scope.

```
body: if owningNamespace = null then resolveGlobal(name)
else
    let memberships : Membership = membership->
        select(memberShortName = name or memberName = name) in
    if memberships->notEmpty() then memberships->first()
    else owningNamspace.resolveLocal(name)
    endif
endif
```

resolveVisible(name : String) : Membership [0..1]

Resolve a simple name from the visible `Memberships` of this `Namespace`.

```
body: let memberships : Sequence(Membership) =
    visibleMemberships(Set{}, false, false)->
    select(memberShortName = name or memberName = name) in
if memberships->isEmpty() then null
else memberships->first()
endif
```

unqualifiedNameOf(qualifiedName : String) : String

Return the simple name that is the last segment name of the given `qualifiedName`. If this segment name has the form of a KerML unrestricted name, then "unescape" it by removing the surrounding single quotes and replacing all escape sequences with the specified character.

```
body: No OCL
```

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->
        select(importedMemberships(Set{})->includes(mem)).
        first().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 visibleMemberships : Sequence(Membership) =
    if includeAll then memberships
    else ownedMembership->
        select(visibility = VisibilityKind::public)->
        union(ownedImport->
            select(visibility = VisibilityKind::public).
            importedMemberships(excluded->including(self)))
    endif in
if not isRecursive then visibleMemberships
else visibleMemberships->union(visibleMemberships->
        selectAsKind(Namespace).
        visibleMemberships(excluded->including(self), true, includeAll))
endif
```

Constraints

deriveNamespaceImportedMembership

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

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

deriveNamespaceMembers

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

```
member = membership.memberElement
```

deriveNamespaceOwnedImport

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

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

deriveNamespaceOwnedMember

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

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

deriveNamespaceOwnedMembership

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

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

validateNamespaceDistinguishibility

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

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

8.3.2.4.6 NamespaceImport

Description

A `NamespaceImport` is an Import that imports `Memberships` from its `importedNamespace` into the `importOwningNamespace`. If `isRecursive = false`, then only the visible `Memberships` of the `importOwningNamespace` are imported. If `isRecursive = true`, then, in addition, `Memberships` are recursively imported from any `ownedMembers` of the `importedNamespace` that are `Namespaces`.

General Classes

Import

Attributes

importedNamespace : Namespace {redefines target}

The `Namespace` whose visible `Memberships` are imported by this `NamespaceImport`.

Operations

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

Returns at least the visible `Memberships` of the `importedNamespace`. If `isRecursive = true`, then `Memberships` are also recursively imported from any `ownedMembers` of the `importedNamespace` that are themselves `Namespaces`.

```
body: if excluded->includes(importedNamespace) then Sequence{}
else importedNamespace.visibleMemberships(excluded, isRecursive, isImportAll)
```

Constraints

deriveNamespaceImportImportedElement

The `importedElement` of a `NamespaceImport` is its `importedNamespace`.

```
importedElement = importedNamespace
```

8.3.2.4.7 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., `Specialization`).

public

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

8.3.2.4.8 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 `name` of the `ownedMemberElement`.

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

The `shortName` of the `ownedMemberElement`.

Operations

None.

Constraints

deriveOwningMembershipOwnedMemberName

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

```
ownedMemberName = ownedMemberElement.name
```

deriveOwningMembershipOwnedMemberShortName

The `ownedMemberShortName` of an OwningMembership is the `shortName` of its `ownedMemberElement`.

```
ownedMemberShortName = ownedMemberElement.shortName
```

8.3.3 Core Abstract Syntax

8.3.3.1 Types Abstract Syntax

8.3.3.1.1 Overview

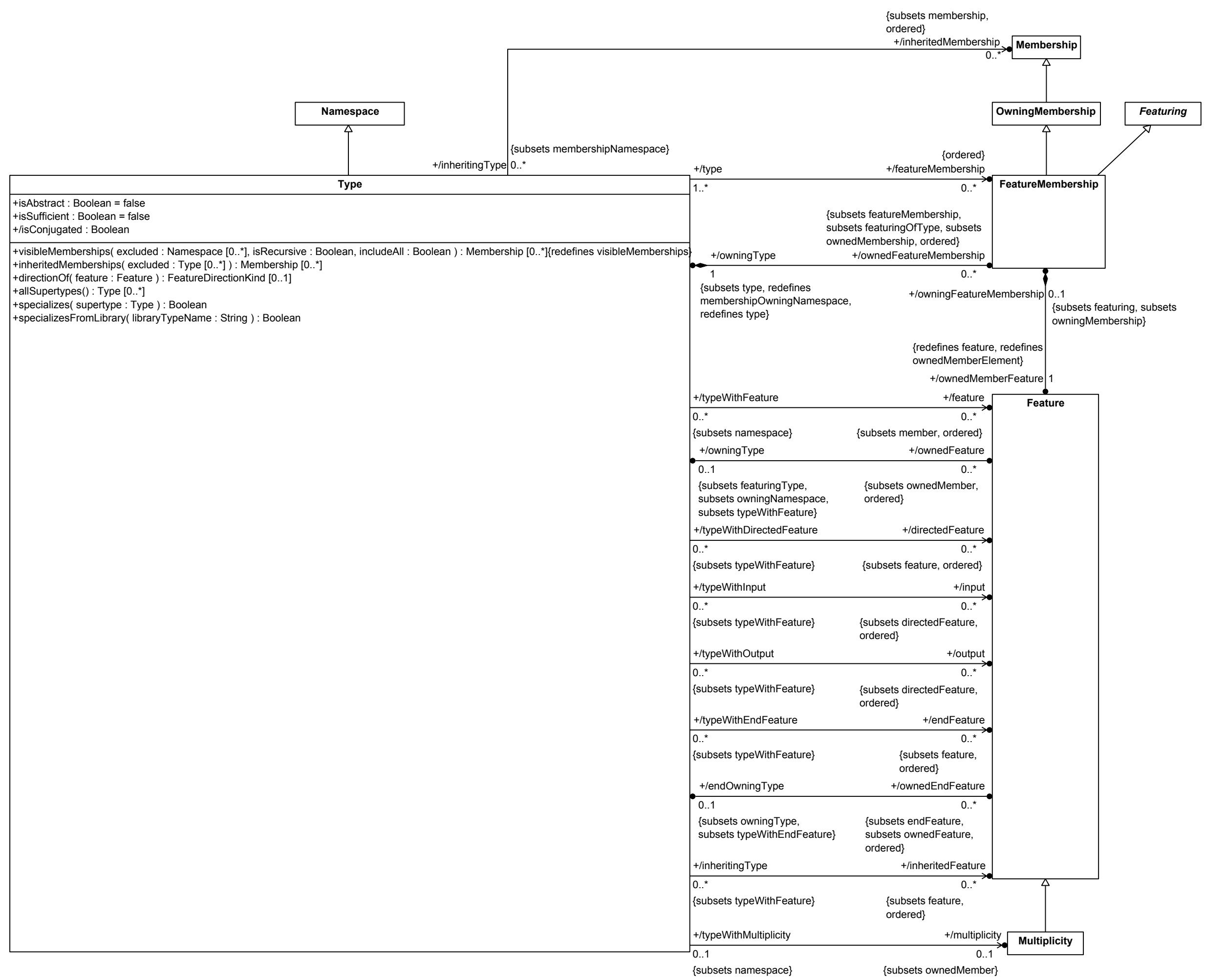

Figure 9. Types

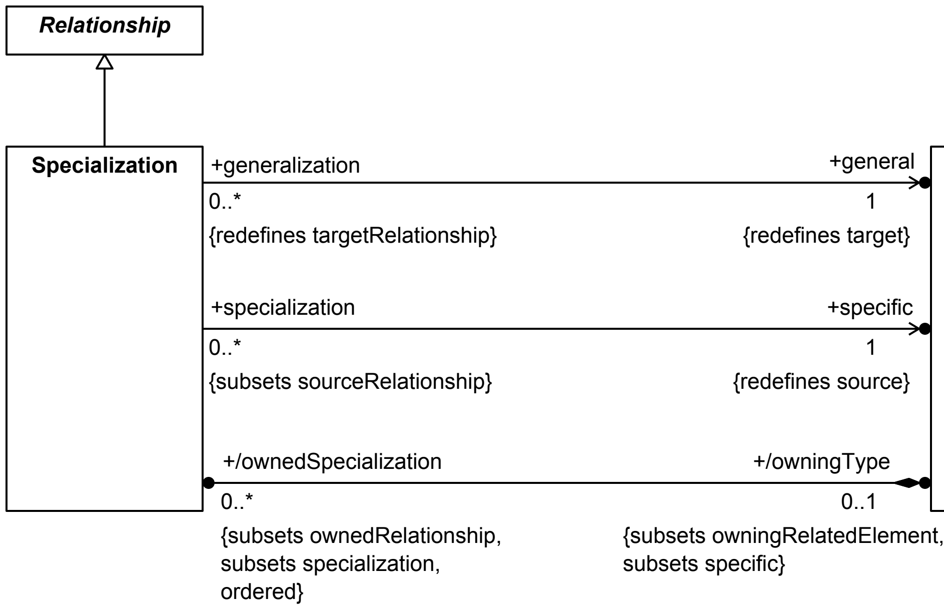

Figure 10. Specialization

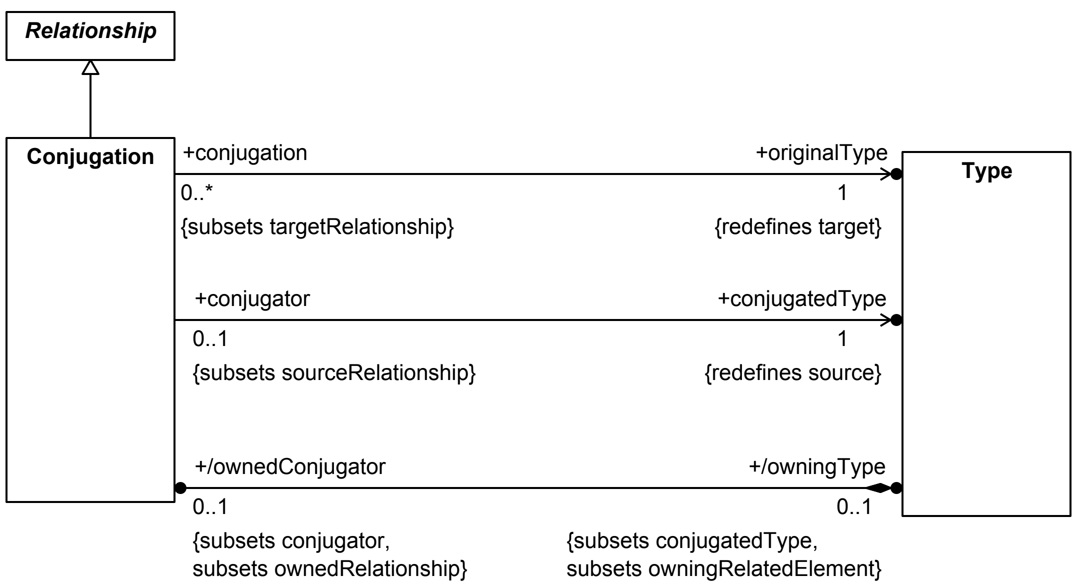

Figure 11. Conjugation

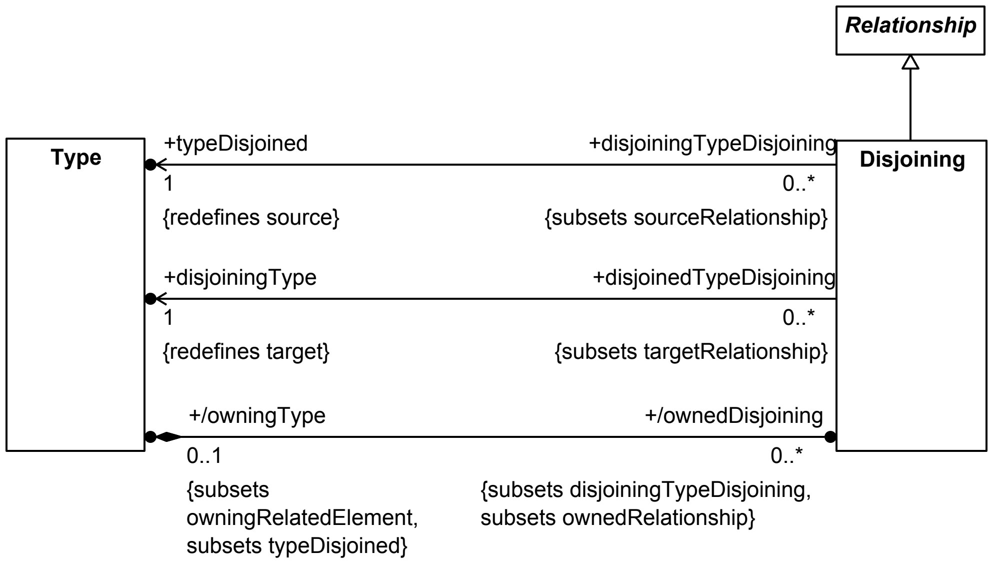

Figure 12. Disjoining

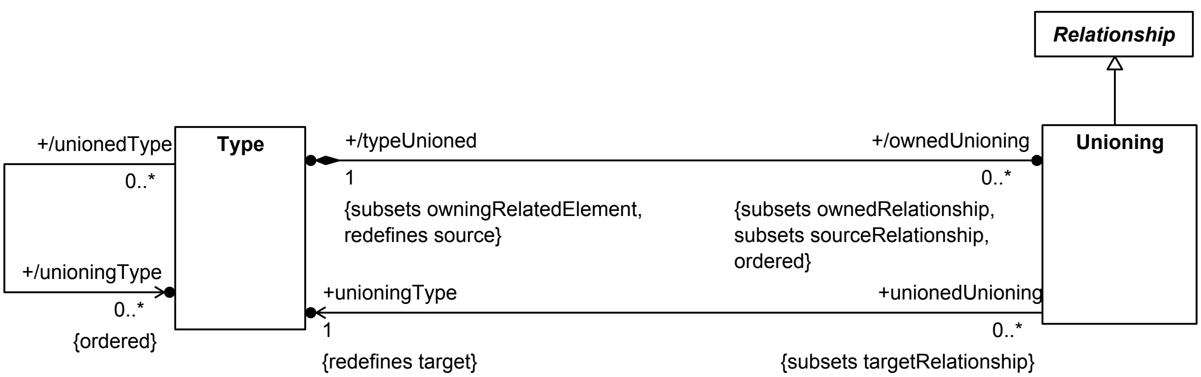

Figure 13. Unioning

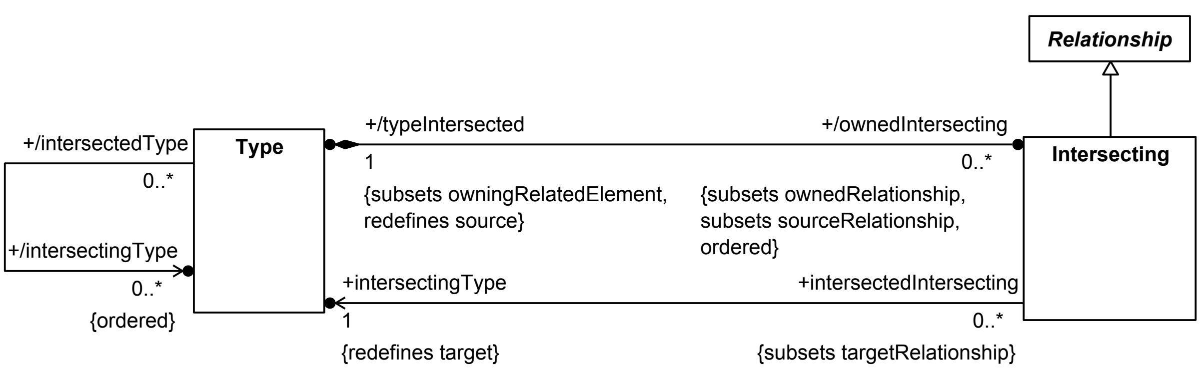

Figure 14. Intersecting

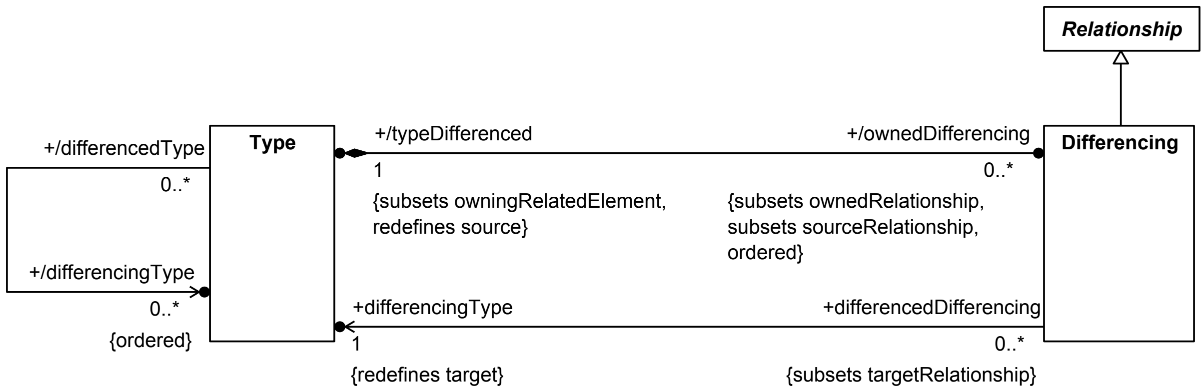

Figure 15. Differencing

8.3.3.1.2 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 `Specialization` 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 `Conjugation` that is also its `owningRelatedElement`.

Operations

None.

Constraints

None.

8.3.3.1.3 Differencing

Description

`Differencing` is a `Relationship` that makes its `differencingType` one of the `differencingTypes` of its `typeDifferenced`.

General Classes

Relationship

Attributes

differencingType : Type {redefines target}

`Type` that partly determines interpretations of `typeDifferenced`, as described in `Type::differencingType`.

/typeDifferenced : Type {subsets owningRelatedElement, redefines source}

`Type` with interpretations partly determined by `differencingType`, as described in `Type::differencingType`.

Operations

None.

Constraints

None.

8.3.3.1.4 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

None.

Constraints

None.

8.3.3.1.5 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.

8.3.3.1.6 FeatureMembership

Description

A `FeatureMembership` is an `OwningMembership` between a `Feature` in an `owningType` that is also a `Featuring` `RelationshipFeature` and the `Type`, in which the `featuringType` is the `source` and the `featureOfType` is the `target`. A `FeatureMembership` is always owned by its `owningType`, which is the `featuringType` for the `FeatureMembership` considered as a `Featuring`.

General Classes

Featuring
OwningMembership

Attributes

/ownedMemberFeature : Feature {redefines ownedMemberElement, feature}

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

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

The `Type` that owns this `FeatureMembership`.

Operations

None.

Constraints

None.

8.3.3.1.7 Intersecting

Description

`Intersecting` is a `Relationship` that makes its `intersectingType` one of the `intersectingTypes` of its `typeIntersected`.

General Classes

Relationship

Attributes

intersectingType : Type {redefines target}

`Type` that partly determines interpretations of `typeIntersected`, as described in `Type::intersectingType`.

/typeIntersected : Type {subsets owningRelatedElement, redefines source}

`Type` with interpretations partly determined by `intersectingType`, as described in `Type::intersectingType`.

Operations

None.

Constraints

None.

8.3.3.1.8 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

None.

Constraints

validateSpecificationSpecificNotConjugated

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

```
not specific.isConjugated
```

8.3.3.1.9 Multiplicity

Description

A `Multiplicity` is a `Feature` whose co-domain is a set of natural numbers giving the allowed cardinalities of each `typeWithMultiplicity`. The *cardinality* of a `Type` is defined as follows, depending on whether the `Type` is a `Classifier` or `Feature`.

- `Classifier` – The number of basic instances of the `Classifier`, that is, those instances representing things, which are not instances of any subtypes of the `Classifier` that are `Features`.
- `Features` – The number of instances with the same featuring instances. In the case of a `Feature` with a `Classifier` as its `featuringType`, this is the number of values of `Feature` for each basic instance of the `Classifier`. Note that, for non-unique `Features`, all duplicate values are included in this count.

`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 *`Base::Anything`*. If the `typeWithMultiplicity` is a `Feature`, the `Multiplicity` shall have the same domain as the `typeWithMultiplicity`.

General Classes

Feature

Attributes

None.

Operations

None.

Constraints

checkMultiplicitySpecialization

A `Multiplicity` must directly or indirectly specialize the `Feature` *`Base::naturals`* from the Kernel Semantic Library.

```
specializesFromLibrary("Base::naturals")
```

checkMultiplicityTypeFeaturing

If the `owningType` of a `Multiplicity` is a `Feature`, then the `Multiplicity` must have the same `featuringTypes` as that `Feature`. Otherwise, it must have no `featuringTypes` (meaning that it is implicitly featured by the base `Classifier` *`Anything`*).

```
if owningType <> null and owningType.oclIsKindOf(Feature) then
    featuringType =
        owningType.oclAsType(Feature).featuringType
else
    featuringType->isEmpty()
endif
```

8.3.3.1.10 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

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

The interpretations of a `Type` with `differencingTypes` are asserted to be those of the first of those `Types`, but not including those of the remaining `Types`. For example, a `Classifier` might be the difference of a `Classifier` for people and another for people of a particular nationality, leaving people who are not of that nationality. Similarly, a feature of people might be the difference between a feature for their children and a `Classifier` for people of a particular sex, identifying their children not of that sex (because the interpretations of the children `Feature` that identify those of that sex are also interpretations of the `Classifier` for that sex).

/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` of this `Type` with `isEnd = true`.

/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 that are FeatureMemberships.

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

All Memberships inherited by this Type via Specialization 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.

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

The interpretations of a Type with intersectingTypes are asserted to be those in common among the intersectingTypes, which are the Types derived from the intersectingType of the ownedIntersectings of this Type. For example, a Classifier might be an intersection of Classifiers for people of a particular sex and of a particular nationality. Similarly, a feature for people's children of a particular sex might be the intersection of a Feature for their children and a Classifier for people of that sex (because the interpretations of the children Feature that identify those of that sex are also interpretations of the Classifier for that sex).

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.

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 classify as cars. However, if the Type *Car* were sufficient, it would classify all four-wheeled things.)

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

An ownedMember of this Type that is a Multiplicity, which constraints the cardinality of the Type. If there is no such ownedMember, then the cardinality of this Type is constrained by all the Multiplicity constraints applicable to any direct supertypes.

/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`.

/ownedDifferencing : Differencing [0..*] {subsets sourceRelationship, ownedRelationship, ordered}

The `ownedRelationships` of this `Type` that are `Differencings`, having this `Type` as their `typeDifferenced`.

/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, featureMembership, featuringOfType, 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`.

/ownedIntersecting : Intersecting [0..*] {subsets ownedRelationship, sourceRelationship, ordered}

The `ownedRelationships` of this `Type` that are `Intersectings`, have the `Type` as their `typeIntersected`.

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

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

/ownedUnioning : Unioning [0..*] {subsets ownedRelationship, sourceRelationship, ordered}

The `ownedRelationships` of this `Type` that are `Unionings`, having the `Type` as their `typeUnioned`.

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

The interpretations of a `Type` with `unioningTypes` are asserted to be the same as those of all the `unioningTypes` together, which are the `Types` derived from the `unioningType` of the `ownedUnionings` of this `Type`. For example, a `Classifier` for people might be the union of `Classifiers` for all the sexes. Similarly, a feature for people's children might be the union of features dividing them in the same ways as people in general.

Operations

allSupertypes() : Type [0..*]

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

```
body: ownedSpecialization->
    closure(general.ownedSpecialization).general->
    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.

specializes(supertype : Type) : Boolean

Check whether this `Type` is a direct or indirect specialization of the given `supertype`.

```
body: if isConjugated then
    ownedConjugator.originalType.specializes(supertype)
else
    allSupertypes()->includes(supertype)
endif
```

specializesFromLibrary(libraryTypeName : String) : Boolean

Check whether this `Type` is a direct or indirect specialization of the named library `Type`. `libraryTypeName` must conform to the syntax of a KerML qualified name and must resolve to a `Type` in global scope.

```
body: let mem : Membership = resolveGlobal(libraryTypeName) in
mem <> null and mem.memberElement.oclIsKindOf(Type) and
specializes(mem.memberElement.oclAsType(Type))
```

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

checkTypeSpecialization

A Type must directly or indirectly specialize *`Base::Anything`* from the Kernel Semantic Library.

```
specializesFromLibrary('Base::Anything')
```

deriveTypeDifferencingType

The `differencingTypes` of a Type are the `differencingTypes` of its `ownedDifferencings`, in the same order.

```
differencingType = ownedDifferencing.differencingType
```

deriveTypeDirectedFeature

The `directedFeatures` of a `Type` are those `features` for which the `direction` is non-null.

```
directedFeature = feature->select(f | directionOf(f) <> null)
```

deriveTypeEndFeature

The `endFeatures` of a `Type` are all its `features` for which `isEnd = true`.

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

deriveTypeFeature

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

```
feature = featureMembership.ownedMemberFeature
```

deriveTypeFeatureMembership

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

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

deriveTypeInheritedFeature

The `inheritedFeatures` of this `Type` are the `memberFeatures` of the `inheritedMemberships` that are `FeatureMemberships`.

```
inheritedFeature = inheritedMemberships->
    selectByKind(FeatureMembership).memberFeature
```

deriveTypeInheritedMembership

The `inheritedMemberships of a Type are determined by the inheritedMemberships() operation.`

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

deriveTypeInput

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

```
input =
    if isConjugated then
        conjugator.originalType.output
    else
```

```
        feature->select(direction = _'in' or direction = inout)
    endif
```

deriveTypeIntersectingType

The intersectingTypes of a Type are the intersectingTypes of its ownedIntersectings.

```
intersectingType = ownedIntersecting.intersectingType
```

deriveTypeMultiplicity

If a Type has an owned Multiplicity, then that is its multiplicity. Otherwise, if the Type has an ownedSpecialization, then its multiplicity is the multiplicity of the general Type of that Specialization.

```
multiplicity =
    let ownedMultiplicities: Sequence(Multiplicity) =
        ownedMember->selectByKind(Multiplicity) in
    if ownedMultiplicities->isEmpty() then null
    else ownedMultiplicities->first()
    endif
```

deriveTypeOutput

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

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

deriveTypeOwnedConjugator

The ownedConjugator of a Type is the its single ownedRelationship that is a Conjugation.

```
let ownedConjugators: Sequence(Conjugator) =
    ownedRelationship->selectByKind(Conjugation) in
    ownedConjugator =
        if ownedConjugators->isEmpty() then null
        else ownedConjugators->at(1) endif
```

deriveTypeOwnedDifferencing

The ownedDifferencings of a Type are its ownedRelationships that are Differencings.

```
ownedRelationship->selectByKind(Differencing)
```

deriveTypeOwnedDisjoining

The ownedDisjoinings of a Type are the ownedRelationships that are Disjoinings.

```
ownedRelationship->selectByKind(Disjoining)
```

deriveTypeOwnedEndFeature

The `ownedEndFeatures` of a `Type` are all its `ownedFeatures` for which `isEnd = true`.

```
ownedEndFeature = ownedFeature->select(isEnd)
```

deriveTypeOwnedFeature

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

```
ownedFeature = ownedFeatureMembership.ownedMemberFeature
```

deriveTypeOwnedFeatureMembership

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

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

deriveTypeOwnedIntersecting

The `ownedIntersectings` of a `Type` are the `ownedRelationships` that are `Intersectings`.

```
ownedRelationship->selectByKind(Intersecting)
```

deriveTypeOwnedSpecialization

The `ownedSpecializations` of a `Type` are the `ownedRelationships` that are `Specializations` whose `special Type` is the owning `Type`.

```
ownedSpecialization = ownedRelationship->selectByKind(Specialization)->
    select(s | s.special = self)
```

deriveTypeOwnedUnioning

The `ownedUnionings` of a `Type` are the `ownedRelationships` that are `Unionings`.

```
ownedRelationship->selectByKind(Unioning)
```

deriveTypeUnioningType

The `unioningTypes` of a `Type` are the `unioningTypes` of its `ownedUnionings`.

```
unioningType = ownedUnioning.unioningType
```

validateTypeAtMostOneConjugator

```
ownedRelationship->selectByKind(Conjugator)->size() <= 1
```

validateTypeDifferencingTypesNotSelf

A Type cannot be one of its own `differencingTypes`.

```
differencingType->excludes(self)
```

validateTypeIntersectingTypesNotSelf

A Type cannot be one of its own `intersectingTypes`.

`intersectingType->excludes(self)`

validateTypeOwnedMultiplicity

A `Type` may have at most one `ownedMember` that is a `Multiplicity`.

```
ownedMember->selectByKind(Multiplicity)->size() <= 1
```

validateTypeUnioningTypesNotSelf

A Type cannot be one of its own `unioningTypes`.

```
unioningType->excludes(self)
```

8.3.3.1.11 Unioning

Description

`Unioning` is a `Relationship` that makes its `unioningType` one of the `unioningTypes` of its `typeUnioned`.

General Classes

Relationship

Attributes

/typeUnioned : Type {subsets owningRelatedElement, redefines source}

`Type` with interpretations partly determined by `unioningType`, as described in `Type::unioningType`.

unioningType : Type {redefines target}

`Type` that partly determines interpretations of `typeUnioned`, as described in `Type::unioningType`.

Operations

None.

Constraints

None.

8.3.3.2 Classifiers Abstract Syntax

8.3.3.2.1 Overview

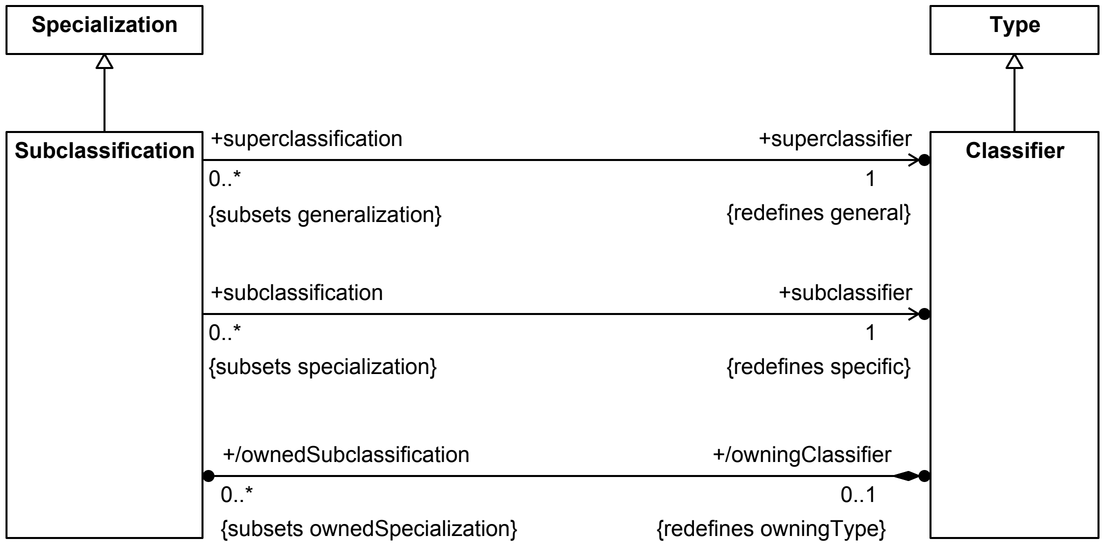

Figure 16. Classifiers

8.3.3.2.2 Classifier

Description

A `Classifier` is a `Type` that classifies:

- Things (in the universe) regardless of how `Features` relate them. (These are interpreted semantically as sequences of exactly one thing.)
- How the above things are related by `Features.` (These are interpreted semantically as sequences of multiple things, such that the last thing in the sequence is also classified by the `Classifier`. Note that his means that a `Classifier` modeled as specializing a `Feature` cannot classify anything.)

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

None.

Constraints

deriveClassifierOwnedSubclassification

The `ownedSubclassifications` of a `Classifier` are its `ownedSpecializations` that are `Subclassifications`.

```
ownedSubclassification =
    ownedSpecialization->selectByKind(Superclassification)
```

validateClassifierMultiplicityDomain

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

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

8.3.3.2.3 Subclassification

Description

`Subclassification` is `Specialization` in which both the `specific` and `general` `Types` are `Classifier`. 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 `Classifier` 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

None.

Constraints

None.

8.3.3.3 Features Abstract Syntax

8.3.3.3.1 Overview

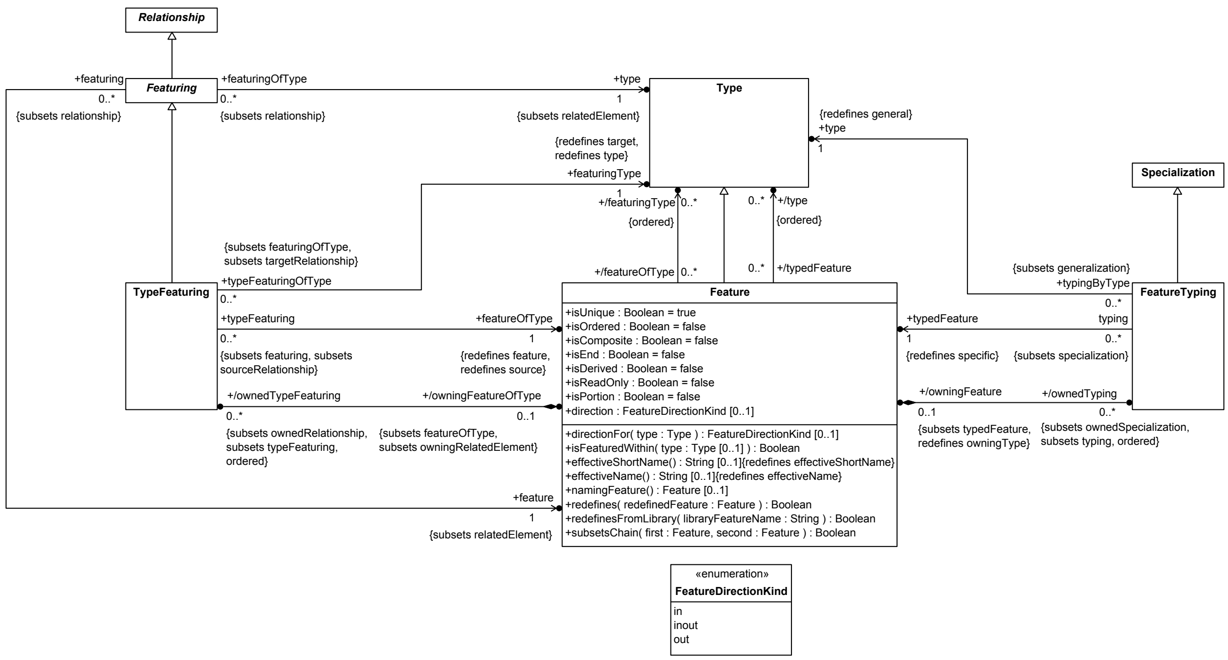

Figure 17. Features

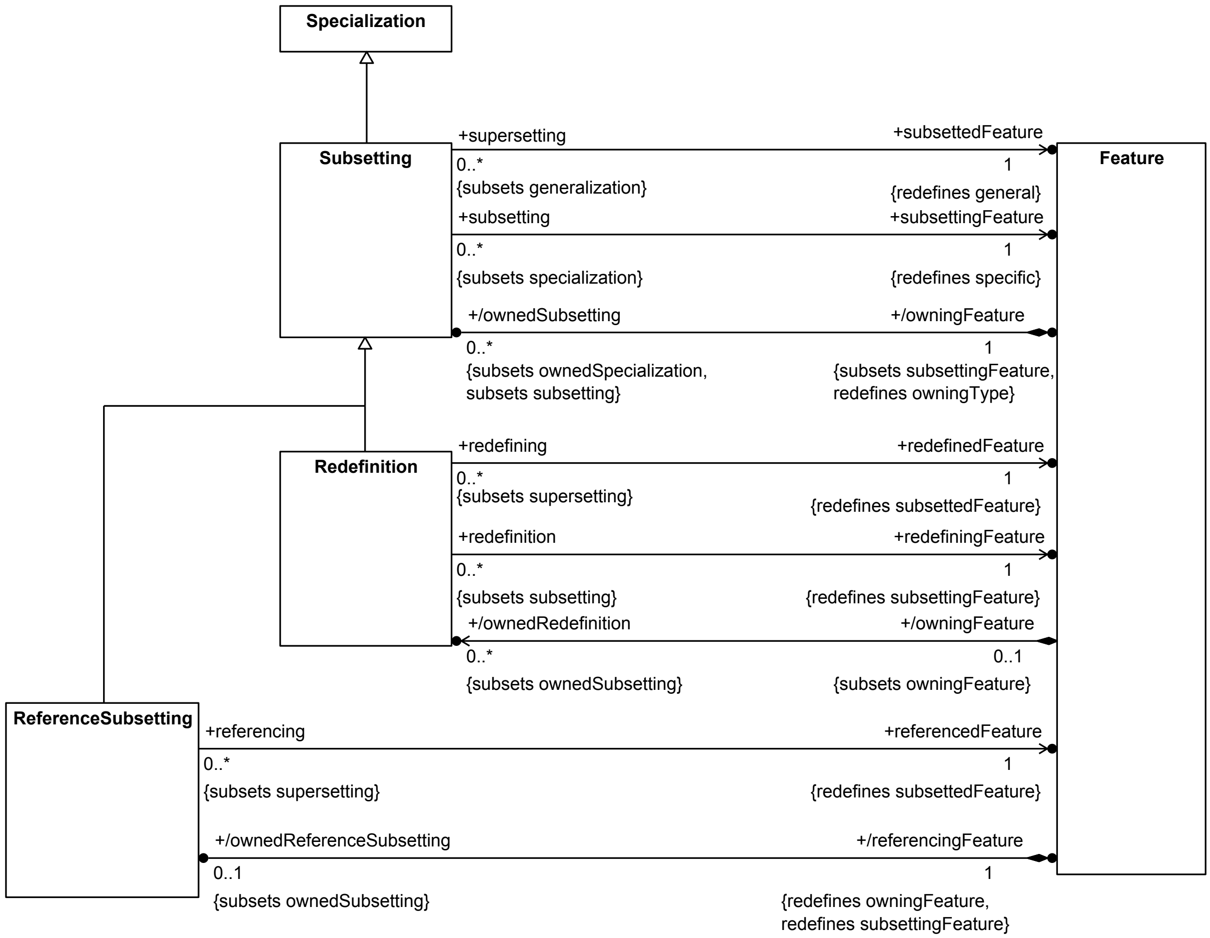

Figure 18. Subsetting

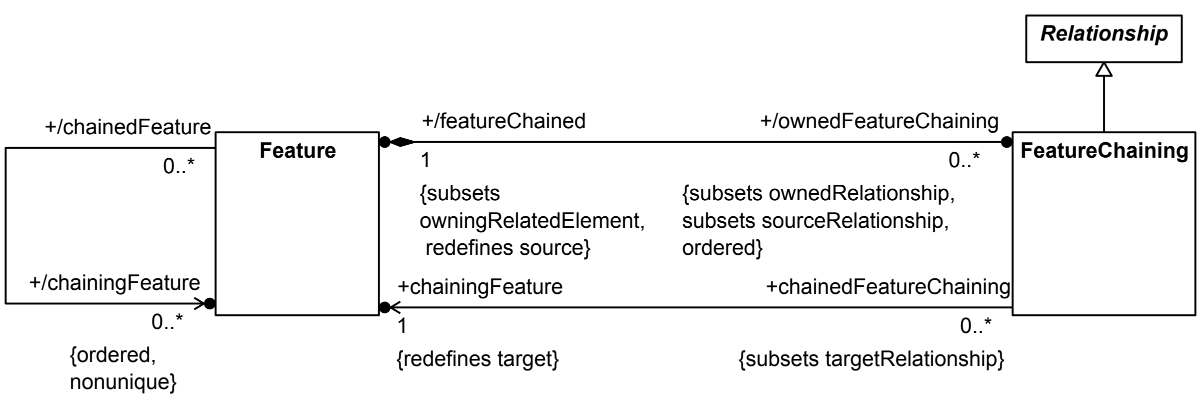

Figure 19. Feature Chaining

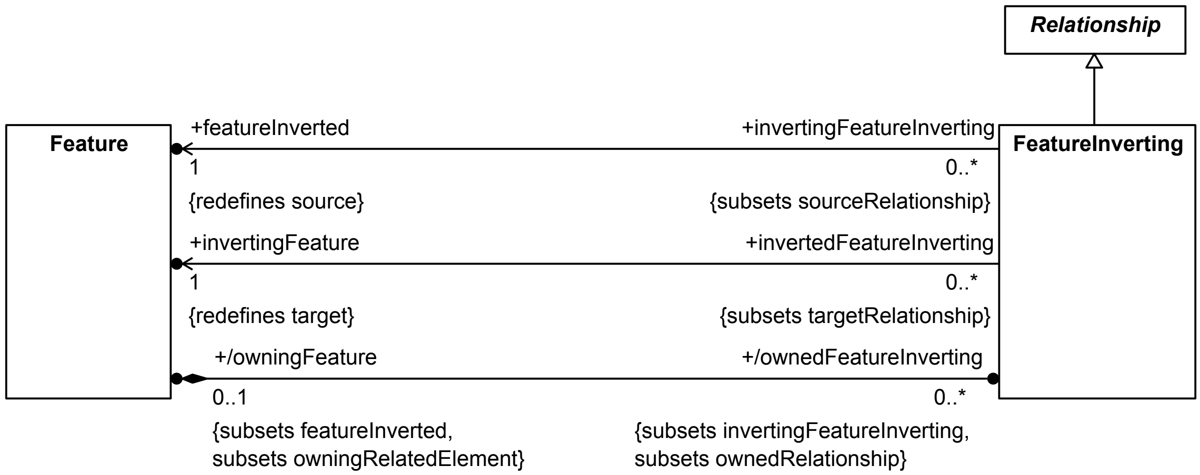

Figure 20. Feature Inverting

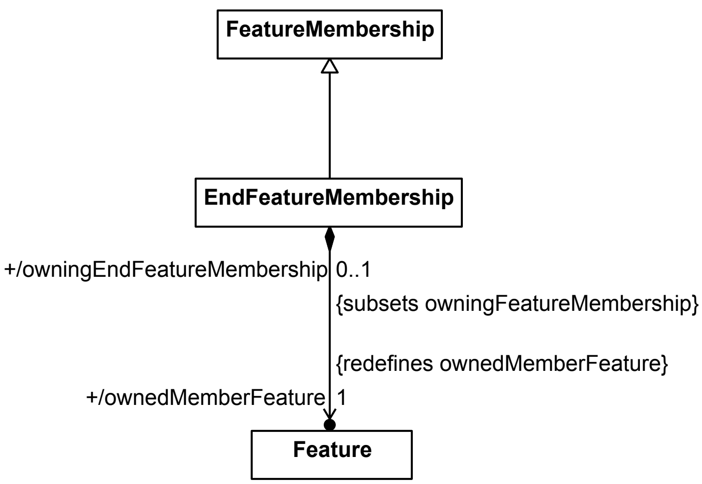

Figure 21. End Feature Membership

8.3.3.3.2 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

None.

Constraints

validateEndFeatureMembershipIsEnd

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

```
ownedMemberFeature.isEnd
```

8.3.3.3.3 Feature

Description

A `Feature` is a `Type` that classifies relations between multiple things (in the universe). The domain of the relation is the intersection of the `featuringTypes` of the `Feature`. (The domain of a `Feature` with no `featuringTyps` is implicitly the most general `Type` *`Base::Anything`* from the Kernel Semantic Library.) The co-domain of the relation is the intersection of the `types` of the `Feature`.

In the simplest cases, the `featuringTypes` and `types` are `Classifiers` and the `Feature` relates two things, one from the domain and one from the range. 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`. In this case, the `Feature` effectively classifies relations between relations, which can be interpreted as the sequence of things related by the domain `Feature` concatenated with the sequence of things related by the co-domain `Feature`.

The *values* of a `Feature` for a given instance of its domain are all the instances of its co-domain that are related to that domain instance by the `Feature`. 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 using those values as domain instances to obtain valus of the second `Feature`, and so on, to values of the last `Feature`.

General Classes

Type

Attributes

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

The `Feature` 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 domain of the original `Feature`), then using each of those as domain instances to find the values of the second `Feature` in chainingFeatures, and so on, to values of the last `Feature`.

direction : FeatureDirectionKind [0..1]

Indicates how values of this `Feature` are determined or used (as specified for the `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 typeFeaturings. If the Feature is chained, then the featuringTypes of the first Feature in the chain are also featuringTypes of the chained Feature.

isComposite : Boolean

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

isDerived : Boolean

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

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 instance to a single co-domain instance, 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 domain of the Feature and represent the same thing as those instances.

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 ownedRelationships of this Feature that are FeatureChainings, for which the Feature will be the featureChained.

/ownedFeatureInverting : FeatureInverting [0..*] {subsets ownedRelationship, invertingFeatureInverting}

The ownedRelationships of this Feature that are FeatureInvertings and for which the Feature is the featureInverted.

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

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

/ownedReferenceSubsetting : ReferenceSubsetting [0..1] {subsets ownedSubsetting}

The one `ownedSubsetting` of this `Feature`, if any, that is a `ReferenceSubsetting`, for which the `Feature` is the `referencingFeature`.

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

The `ownedSpecializations` 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` and for which the `Feature` is the `featureOfType`.

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

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

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

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 `Feature`.

/type : Type [0..*] {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 `typings` and the `types` of its `subsettings`. If the `Feature` is chained, then the `types` of the last `Feature` in the chain are also `types` of the chained `Feature`.

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 `declaredName` or `declaredShortName`, then its effective `name` is given by the effective `name` of the `Feature` returned by the `namingFeature()` operation, if any.

```
body: if declaredShortName <> null or declaredName <> null then
    declaredName
else
```

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

effectiveShortName() : String [0..1]

If a `Feature` has no `declaredShortName` or `declaredName`, then its effective `shortName` is given by the effective `shortName` of the `Feature` returned by the `namingFeature()` operation, if any.

```
body: if declaredShortName <> null or declaredName <> null then
    declaredShortName
else
    let namingFeature : Feature = namingFeature() in
    if namingFeature = null then
        null
    else
        namingFeature.effectiveShortName()
    endif
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` of its first `ownedRedefinition`, if any.

```
body: if ownedRedefinition->isEmpty() then
    null
else
    ownedRedefinition->at(1).redefinedFeature
endif
```

redefines(redefinedFeature : Feature) : Boolean

Check whether this `Feature` *directly* redefines the given `redefinedFeature`.

```
body: ownedRedefinition.redefinedFeature->includes(redefinedFeature)
```

redefinesFromLibrary(libraryFeatureName : String) : Boolean

Check whether this `Feature` *directly* redefines the named library `Feature`. `libraryFeatureName` must conform to the syntax of a KerML qualified name and must resolve to a `Feature` in global scope.

```
body: let mem: Membership = resolveGlobal(libraryFeatureName) in
mem <> null and mem.memberElement.oclIsKindOf(Feature) and
redefines(mem.memberElement.oclAsType(Feature))
```

subsetsChain(first : Feature, second : Feature) : Boolean

Check whether this `Feature` directly or indirectly specializes a `Feature` whose last two `chainingFeatures` are the given `Features` `first` and `second`.

```
body: allSuperTypes()->selectAsKind(Feature)->
    exists(f | let n: Integer = f.chainingFeature->size() in
        n >= 2 and
        f.chainingFeature->at(n-1) = first and
        f.chainingFeature->at(n) = second)
```

Constraints

checkFeatureDataValueSpecialization

If a Feature has an `ownedTyping` relationship to a `DataType`, then it must directly or indirectly specialize *`Base::dataValues`* from the Kernel Semantic Library.

```
ownedTyping.type->exists(selectByKind(DataType)) implies
    specializesFromLibary("Base::dataValues")
```

checkFeatureEndRedefinition

If a `Feature` has `isEnd = true` and an `owningType` that is not empty, then, for each direct supertype of its `owningType`, it must redefine the `endFeature` at the same position, if any.

```
isEnd and owningType <> null implies
    let i : Integer =
        owningType.ownedFeature->select(isEnd) in
    owningType.ownedSpecialization.general->
        forAll(supertype |
            let ownedEndFeatures : Sequence(Feature) =
                supertype.ownedFeature->select(isEnd) in
            ownedEndFeatures->size() >= i implies
                redefines(ownedEndFeatures->at(i))
```

checkFeatureEndSpecialization

If a `Feature` has `isEnd = true` and an `owningType` that is an `Association`, then it must directly or indirectly specialize *`Links::Link::participants`* from the Kernel Semantic Library.

```
isEnd and owningType <> null and
owningType.oclIsKindOf(Association) implies
    specializesFromLibrary("Links::Link::participants")
```

checkFeatureItemFlowFeatureRedefinition

If a `Feature` is the first `ownedFeature` of a first or second `ItemFlowEnd`, then it must directly or indirectly specialize either *`Transfers::Transfer::source::sourceOutput`* or *`Transfers::Transfer::target::targetInput`*, respectively, from the Kernel Semantic Library.

```
owningType <> null and
owningType.oclIsKindOf(ItemFlowEnd) and
```

```
owningType.ownedFeature->at(1) = self implies
    let flowType : Type = owningType.owningType in
    flowType <> null implies
        let i : Integer =
            flowType.ownedFeature.indexOf(owningType) in
        (i = 1 implies
            redefinesFromLibrary("Transfers::Transfer::source::sourceOutput")) and
        (i = 2 implies
            redefinesFromLibrary("Transfers::Transfer::source::targetInput"))
```

checkFeatureObjectSpecialization

If a Feature has an `ownedTyping` relationship to a `Structure`, then it must directly or indirectly specialize *`Objects::objects`* from the Kernel Semantics Library.

```
ownedTyping.type->exists(selectByKind(Structure)) implies
    specializesFromLibary("Objects::objects")
```

checkFeatureOccurrenceSpecialization

If a Feature has an `ownedTyping` relationship to a `Class`, then it must directly or indirectly specialize *`Occurrences::occurrences`* from the Kernel Semantic Library.

```
ownedTyping.type->exists(selectByKind(Class)) implies
    specializesFromLibrary("Occurrences::occurrences")
```

checkFeatureParameterRedefinition

If a `Feature` is a `parameter` of an `owningType` that is a `Behavior` or `Step`, *other* than the `result parameter` (if any), then, for each direct supertype of its `owningType` that is also a `Behavior` or `Step`, it must redefine the `parameter` at the same position, if any.

```
owningType <> null and
(owningType.oclIsKindOf(Behavior) or
 owningType.oclIsKindOf(Step)) implies
    let i : Integer =
        owningType.ownedFeature->select(direction <> null) in
    owningType.ownedSpecialization.general->
        forAll(supertype |
            let ownedParameters : Sequence(Feature) =
                supertype.ownedFeature->select(direction <> null) in
            ownedParameters->size() >= i implies
                redefines(ownedParameters->at(i))
```

checkFeatureResultRedefinition

If a `Feature` is a `result parameter` of an `owningType` that is a `Function` or `Expression`, then, for each direct supertype of its `owningType` that is also a `Function` or `Expression`, it must redefine the `result parameter`.

```
owningType <> null and
(owningType.oclIsKindOf(Function) and
    self = owningType.oclAsType(Function).result or
 owningType.oclIsKindOf(Expression) and
    self = owningType.oclAsType(Expression).result) implies
    owningType.ownedSpecialization.general->
        select(oclIsKindOf(Function) or oclIsKindOf(Expression))->
```

```
        forAll(supertype |
            redefines(
                if superType.oclIsKindOf(Function) then
                    superType.oclAsType(Function).result
                else
                    superType.oclAsType(Expression).result
                endif)
```

checkFeatureResultSpecialization

If a `Feature` has an `owningType` that is a `LiteralExpression` it must directly or indirectly specialize the `DataType` from the *`ScalarValues`* package in the Kernel Data Types Library corresponding to the kind of `LiteralExpression`. If the `owningType` is a `FeatureReferenceExpression`, then the `Feature` must specialize the `referent` of the `FeatureReferenceExpression`.

```
owningType <> null and
(owningType.oclIsKindOf(LiteralExpression) or
 owningType.oclIsKindOf(FeatureReferenceExpression)) implies
    if owningType.oclIsKindOf(LiteralString) then
        specializesFromLibrary("ScalarValues::String")
    else if owningType.oclIsKindOf(LiteralBoolean) then
        specializesFromLibrary("ScalarValues::Boolean")
    else if owningType.oclIsKindOf(LiteralInteger) then
        specializesFromLibrary("ScalarValues::Rational")
    else if owningType.oclIsKindOf(LiteralBoolean) then
        specializesFromLibrary("ScalarValues::Rational")
    else if owningType.oclIsKindOf(LiteralBoolean) then
        specializesFromLibrary("ScalarValues::Real")
    else specializes(
        owningType.oclAsType(FeatureReferenceExpression).referent)
    endif endif endif endif endif
```

checkFeatureSpecialization

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

```
specializesFromLibrary("Base::things")
```

checkFeatureSubobjectSpecialization

A composite `Feature typed by a Structure, and whose ownedType is a Structure or another Feature typed by a Structure must directly or indirectly specialize` *`Objects::Object::subobjects`*.

```
isComposite and
ownedTyping.type->includes(oclIsKindOf(Structure)) and
owningType <> null and
(owningType.oclIsKindOf(Structure) or
 owningType.type->includes(oclIsKindOf(Structure))) implies
    specializesFromLibrary("Occurrence::Occurrence::suboccurrences")
```

checkFeatureSuboccurrenceSpecialization

A composite `Feature that has an ownedTyping relationship to a Class, and whose ownedType is a Class or another Feature typed by a Class, must directly or indirectly specialize` *`Occurrences::Occurrence::suboccurrences`*.

```
isComposite and
ownedTyping.type->includes(oclIsKindOf(Class)) and
owningType <> null and
(owningType.oclIsKindOf(Class) or
 owningType.oclIsKindOf(Feature) and
    owningType.oclAsType(Feature).type->
        exists(oclIsKindOf(Class))) implies
    specializesFromLibrary("Occurrence::Occurrence::suboccurrences")
```

checkFeatureValuationSpecialization

If a `Feature` has a `FeatureValue`, then it must specialize the `result` of the `value` `Expression` of the `FeatureValue`.

```
ownedMembership->
    selectByKind(FeatureValue)->
    forAll(fv | specializes(fv.value.result))
```

deriveFeatureChainingFeature

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

```
chainingFeature = ownedFeatureChaining.chainingFeature
```

deriveFeatureFeaturingType

```
featuringType =
    let featuringTypes : OrderedSet(Type) =
        typeFeaturing.featuringType->asOrderedSet() in
    if chainingFeature->isEmpty() then featuringTypes
    else
        featuringTypes->
            union(chainingFeature->first().featuringType)->
            asOrderedSet()
    endif
```

deriveFeatureOwnedFeatureChaining

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

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

deriveFeatureOwnedFeatureInverting

The `ownedFeatureInvertings` of a `Feature` are its `ownedRelationships` that are `FeatureInvertings`.

```
ownedFeatureInverting = ownedRelationship->selectByKind(FeatureInverting)->
    select(fi | fi.featureInverted = self)
```

deriveFeatureOwnedRedefinition

The `ownedRedefinitions` of a `Feature` are its `ownedSubsettings` that are `Redefinitions`.

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

deriveFeatureOwnedReferenceSubsetting

The `ownedReferenceSubsetting` of a `Feature` is the first `ownedSubsetting` that is a `ReferenceSubsetting` (if any).

```
ownedReferenceSubsetting =
    let referenceSubsettings : OrderedSet(ReferenceSubsetting) =
        ownedSubsetting->selectByKind(ReferenceSubsetting) in
    if referenceSubsettings->isEmpty() then null
    else referenceSubsettings->first() endif
```

deriveFeatureOwnedSubsetting

The `ownedRedefinitions` of a `Feature` are its `ownedSpecializations` that are `Subsettings`.

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

deriveFeatureOwnedTypeFeaturing

The `ownedTypeFeaturings` of a `Feature` are its `ownedRelationships` that are `TypeFeaturings` and which have the `Feature` as their `featureOfType`.

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

deriveFeatureOwnedTyping

The `ownedTypings` of a `Feature` are its `ownedSpecializations` that are `FeatureTypings`.

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

deriveFeatureType

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 `typings` and the `types` of the `subsettedFeatures` of its `subsettings`, with all redundant supertypes removed.

```
type =
    let types : OrderedSet(Type) = typing.type->
        union(subsetting.subsettedFeature.type)->
        asOrderedSet() in
    if chainingFeature->isEmpty() then types
    else
        types->union(chainingFeature->last().type)->
        asOrderedSet()
    endif
```

validateFeatureChainingFeatureNotOne

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

validateFeatureChainingFeaturesNotSelf

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

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

validateFeatureMultiplicityDomain

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
```

validateFeatureOwnedReferenceSubsetting

A `Feature` must have at most one `ownedSubsetting` that is an `ReferenceSubsetting`.

```
ownedSubsetting->selectByKind(ReferenceSubsetting)->size() <= 1
```

8.3.3.3.4 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 {subsets owningRelatedElement, redefines source}

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

Operations

None.

Constraints

None.

8.3.3.3.5 FeatureInverting

Description

A `FeatureInverting` is a `Relationship` between `Features` asserting that their interpretations (sequences) are the reverse of each other, identified as `featureInverted` and `invertingFeature`. For example, a `Feature` identifying each person's parents is the inverse of a `Feature` identifying each person's children. A person identified as a parent of another will identify that other as one of their children.

General Classes

Relationship

Attributes

featureInverted : Feature {redefines source}

The `Feature` that is an inverse of the `invertingFeature`.

invertingFeature : Feature {redefines target}

The `Feature` that is an inverse of the `invertedFeature`.

/owningFeature : Feature [0..1] {subsets owningRelatedElement, featureInverted}

A `featureInverted` that is also the `owningRelatedElement` of this `FeatureInverting`.

Operations

None.

Constraints

None.

8.3.3.3.6 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` has values that are instances of the `Classifier`.

General Classes

Specialization

Attributes

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

A `typedFeature` that is also the `owningRelatedElement` of this `FeatureTyping`.

type : Type {redefines general}

The `Type` that is being applied by this `FeatureTyping`.

typedFeature : Feature {redefines specific}

The `Feature` that has a `type` determined by this `FeatureTyping`.

Operations

None.

Constraints

None.

8.3.3.3.7 Featuring

Description

`Featuring` is a `Relationship` between a `Type` and a `Feature` that is featured by that `Type`. It asserts that every instance in the domain of the `feature` must be classified by the `type`.

`Featuring` is abstract and does not commit to which of `feature` or `type` are the `source` or `target` of the `Relationship`. This commitment is made in the subclasses of `Featuring`, `TypeFeaturing` and `FeatureMembership`, which have opposite directions.

General Classes

Relationship

Attributes

feature : Feature {subsets relatedElement}

The `Feature` that is featured by the `featuringType`.

type : Type {subsets relatedElement}

The `Type that features the featureOfType.`

Operations

None.

Constraints

None.

8.3.3.3.8 Redefinition

Description

`Redefinition` is a kind of `Subsetting` that requires 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 domain of the `redefiningFeature`), and vice versa. The `redefinedFeature` might have values for instances of the domain of the `redefiningFeature`, but only as instances of the domain of the `redefinedFeature` that happen to also be instances of the domain 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 inherited 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

None.

Constraints

validateRedefinitionFeaturingTypes

The `subsettingFeature` of a `Subsetting` must have at least one `featuringType` that is not also a `featuringType` of the `subsettedFeature`.

```
let anythingType: Type =
    subsettingFeature.resolveGlobal('Base::Anything').oclAsType(Type) in
-- Including "Anything" accounts for implicit featuringType of Features
-- with no explicit featuringType.
let subsettingFeaturingTypes: Set(Type) =
    subsettingFeature.featuringTypes->asSet()->including(anythingType) in
let subsettedFeaturingTypes: Set(Type) =
    subsettedFeature.featuringTypes->asSet()->including(anythingType) in
subsettingFeaturingTypes <> subsettedFeaturingType
```

8.3.3.3.9 ReferenceSubsetting

Description

`ReferenceSubsetting` is a kind of `Subsetting` in which the `referencedFeature` is syntactically distinguished from other `Features` subsetted by the `referencingFeature`. `ReferenceSubsetting` has the same semantics as `Subsetting`, but the `referenceFeature` may have a special purpose relative to the `referencingFeature`. For instance, `ReferenceSubsetting` is used to identify the `relatedFeatures` of a `Connector`.

`ReferenceSubsetting` is always an `ownedRelationship` of its `referencingFeature`. A `Feature` can have at most one `ownedReferenceSubsetting`.

General Classes

Subsetting

Attributes

referencedFeature : Feature {redefines subsettedFeature}

The `Feature` that is referenced by the `referencingFeature` of this `ReferenceSubsetting`.

/referencingFeature : Feature {redefines subsettingFeature, owningFeature}

The `Feature` that owns this `ReferenceSubsetting` relationship, which is also its `subsettingFeature`.

Operations

None.

Constraints

None.

8.3.3.3.10 Subsetting

Description

`Subsetting` is `Specialization` in which the `specific` and `general` `Types` 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 `Specialization`), and the co-domain (intersection of the `types`) of the `subsettingFeature` must specialize the co-domain of the `subsettedFeature`.

General Classes

Specialization

Attributes

/owningFeature : Feature {subsets subsettingFeature, redefines owningType}

A `subsettingFeature` that is also the `owningRelatedElement` of this `Subsetting`.

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

None.

Constraints

validateSubsettingFeaturingTypes

For each `featuringType` of the `subsettedFeature` of a `Subsetting`, there must be a `featuringType` of the `subsettingFeature` that conforms to it.

```
let subsettingFeaturingTypes: OrderedSet(Type) =
    subsettingFeature.featuringTypes in
let subsettedFeaturingTypes: OrderedSet(Type) =
    subsettedFeature.featuringTypes in
let anythingType: Element =
    subsettingFeature.resolveGlobal('Base::Anything') in
subsettedFeaturingTypes->forAll(t |
```

```
subsettingFeaturingTypes->isEmpty() and t = anythingType or
subsettingFeaturingTypes->exists(specializes(t))
```

8.3.3.3.11 TypeFeaturing

Description

A `TypeFeaturing` is a `Featuring Relationship` in which the `featureOfType` is the `source` and the `featuringType` is the `target`.

General Classes

Featuring

Attributes

featureOfType : Feature {redefines source, feature}

The `Feature` that is featured by the `featuringType`. It is the `source` of the `TypeFeaturing`.

featuringType : Type {redefines target, type}

The `Type` that features the `featureOfType`. It is the `target` of the `TypeFeaturing`.

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

A `featureOfType` that is also the `owningRelatedElement` of this `TypeFeaturing`.

Operations

None.

Constraints

None.

8.3.4 Kernel Abstract Syntax

8.3.4.1 Data Types Abstract Syntax

8.3.4.1.1 Overview

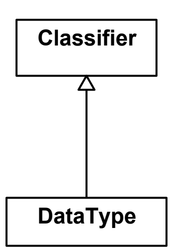

Figure 22. Data Types

8.3.4.1.2 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 different things.
- 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

None.

Constraints

checkDataTypeSpecialization

A `DataType` must directly or indirectly specialize the base `DataType` *`Base::DataValue`* from the Kernel Semantic Library.

```
specializesFromLibrary('Base::DataValue')
```

validateDatatypeSpecialization

A `DataType` must not specialize a `Class` or an `Association`.

```
ownedSpecialization.general->
    forAll(not oclIsKindOf(Class) and
           not oclIsKindOf(Association))
```

8.3.4.2 Classes Abstract Syntax

8.3.4.2.1 Overview

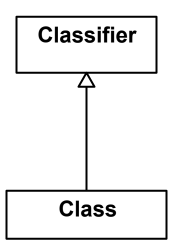

Figure 23. Classes

8.3.4.2.2 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

None.

Constraints

checkClassSpecialization

A `Class` must directly or indirectly specialize the base `Class` *`Occurrences::Occurrence`* from the Kernel Semantic Library.

```
specializesFromLibrary('Occurrences::Occurrence')
```

validateClassSpecialization

A `Class` must not specialize a `DataType` and it can only specialize an `Association` if it is an `AssociationStructure`.

```
ownedSpecialization.general->
    forAll(not oclIsKindOf(DataType)) and
not oclIsKindOf(AssociationStructure) implies
    ownedSpecialization.general->
        forAll(not oclIsKindOf(Association))
```

8.3.4.3 Structures Abstract Syntax

8.3.4.3.1 Overview

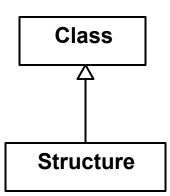

Figure 24. Structures

8.3.4.3.2 Structure

Description

A `Structure` is a `Class` of objects in the modeled universe that are primarily structural in nature. While such 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

None.

Constraints

checkStructureSpecialization

A Structure must directly or indirectly specialize the base Structure `Objects::Object` from the Kernel Semantic Library.

```
specializesFromLibrary('Objects::Object')
```

8.3.4.4 Associations Abstract Syntax

8.3.4.4.1 Overview

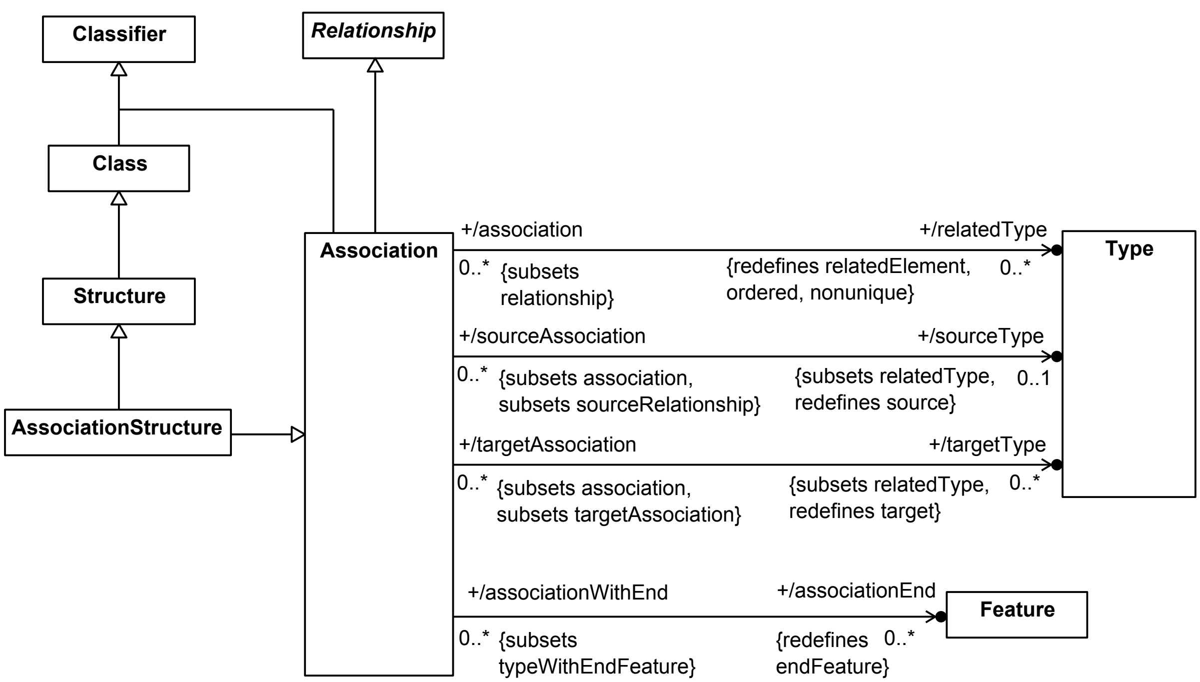

Figure 25. Associations

8.3.4.4.2 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 [0..*] {redefines endFeature}

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

/relatedType : Type [0..*] {redefines relatedElement, ordered, nonunique}

The `types` of the `associationEnds` 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`. It is the first `relatedType` of the `Association`.

/targetType : Type [0..*] {subsets relatedType, redefines target}

The target `relatedTypes` for this `Association`. This includes all the `relatedTypes` other than the `sourceType`.

Operations

None.

Constraints

checkAssociationBinarySpecialization

A binary Association must directly or indirectly specialize the base Association *`Links::binaryLink`* from the Kernel Semantic Library.

```
ownedEndFeature->size() = 2 implies
    specializesFromLibrary("Links::BinaryLink)
```

checkAssociationSpecialization

An Association must directly or indirectly specialize the base Association *`Links::Link`* from the Kernel Semantic Library.

```
specializesFromLibrary("Links::Link")
```

deriveAssociationRelatedType

The `relatedTypes` of an `Association` are the `types` of its `associationEnds`.

```
relatedType = associationEnd.type
```

deriveAssociationSourceType

The `sourceType` of an `Association` is its first `relatedType` (if any).

```
sourceType =
    if relatedType->isEmpty() then null
    else relatedType->first() endif
```

deriveAssociationTargetType

```
targetType =
    if relatedType->size() < 2 then OrderedSet{}
    else
        relatedType->
            subSequence(2, relatedType->size())->
            asOrderedSet()
    endif
```

validateAssociationBinarySpecialization

If an `Association` has more than two `associationEnds`, then it must *not* specialize, directly or indirectly, the `Association` *`BinaryLink`* from the Kernel Semantic Library.

```
associationEnds->size() > 2 implies
    not specializesFromLibrary("Links::BinaryLink")
```

validateAssociationRelatedTypes

If an Association is concrete (not abstract), then it must have at least two `relatedTypes`.

```
not isAbstract implies relatedType->size() >= 2
```

validateAssociationStructureIntersection

If an `Association` is also a kind of `Structure`, then it must be an `AssociationStructure`.

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

8.3.4.4.3 AssociationStructure

Description

An `AssociationStructure` is an `Association` that is also a `Structure`, classifying link objects that are both links and objects. As objects, link objects can be created and destroyed, and their non-end `Features` can change over time. However, the values of the end `Features` of a link object are fixed and cannot change over its lifetime.

General Classes

Structure
Association

Attributes

None.

Operations

None.

Constraints

checkAssociationStructureBinarySpecialization

A binary AssociationStructure must directly or indirectly specialize the base AssociationStructure *`Objects::BinaryLinkObject`* from the Kernel Semantic Library.

```
endFeature->size() = 2 implies
    specializesFromLibrary("Objects::BinaryLinkObject")
```

checkAssociationStructureSpecialization

An AssociationStructure must directly or indirectly specialize the base AssociationStructure *`Objects::LinkObject`* from the Kernel Semantic Library.

```
specializesFromLibrary("Objects::ObjectLink")
```

8.3.4.5 Connectors Abstract Syntax

8.3.4.5.1 Overview

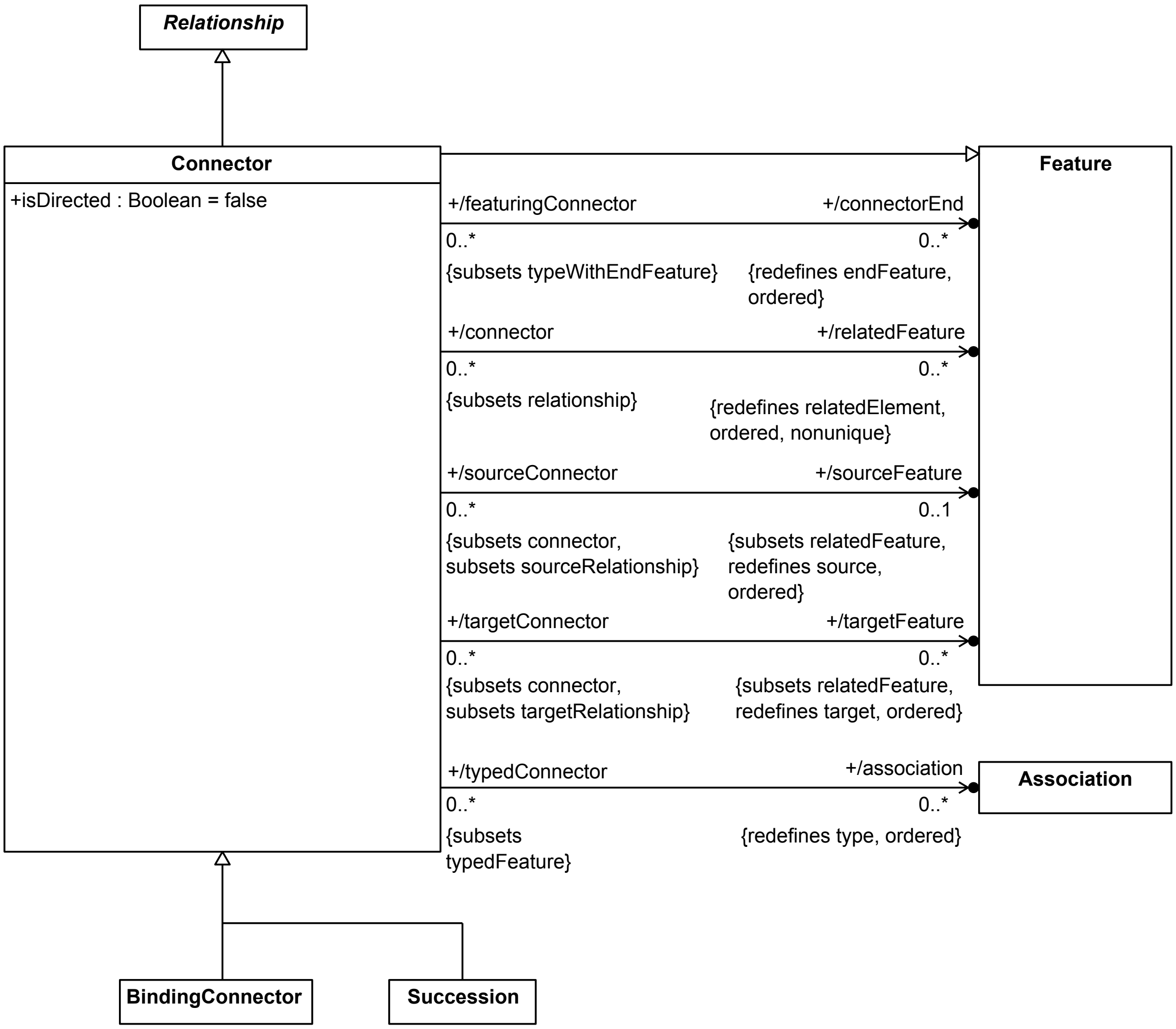

Figure 26. Connectors

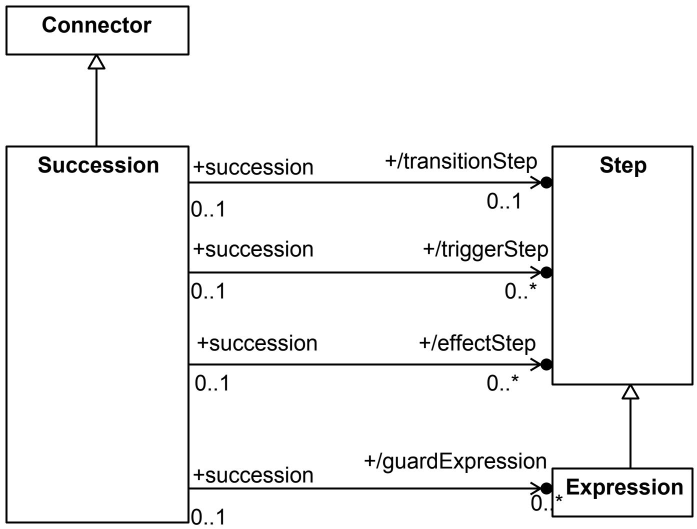

Figure 27. Successions

8.3.4.5.2 Binding Connector

Description

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

General Classes

Connector

Attributes

None.

Operations

None.

Constraints

checkBindingConnectorSpecialization

A BindingConnector must directly or indirectly specialize the base BindingConnector *`Links::selfLinks`* from the Kernel Semantic Library.

```
specializesFromLibrary("Links::selfLinks")
```

validateBindingConnectorIsBinary

A `BindingConnector` must be binary.

```
relatedFeature->size() = 2
```

8.3.4.5.3 Connector

Description

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

General Classes

Relationship
Feature

Attributes

/association : Association [0..*] {redefines type, ordered}

The `Associations` that type the `Connector`.

/connectorEnd : Feature [0..*] {redefines endFeature, ordered}

The `endFeatures` of a `Connector`, which redefine the `endFeatures` of the `associations` of the `Connector`. The `connectorEnds` determine via `ReferenceSubsetting` `Relationships` which `Features` are related by the `Connector`.

isDirected : Boolean

For a binary `Connector`, whether or not the `Connector` should be considered to have a direction from `sourceFeature` to `targetFeature`.

/relatedFeature : Feature [0..*] {redefines relatedElement, ordered, nonunique}

The `Features` that are related by this `Connector` considered as a `Relationship` and that restrict the links it identifies, given by the referenced `Features` of the `connectorEnds` of the `Connector`.

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

The source `relatedFeature` for this `Connector`. It is the first `relatedFeature`.

/targetFeature : Feature [0..*] {subsets relatedFeature, redefines target, ordered}

p>The target `relatedFeatures` for this `Connector`. This includes all the `relatedFeatures` other than the `sourceFeature`.

Operations

None.

Constraints

checkConnectorBinaryObjectSpecialization

A binary `Connector` for an `AssociationStructure` must directly or indirectly specialize the base `Connector` *`Objects::binaryLinkObjects`* from the Kernel Semantic Library.

```
connectorEnds->size() = 2 and
association->exists(oclIsKindOf(AssocationStructure)) implies
    specializesFromLibrary("Objects::binaryLinkObjects")
```

checkConnectorBinarySpecialization

A binary `Connector` must directly or indirectly specialize the base `Connector` *`Links::binaryLinks`* from the Kernel Semantic Library.

```
connectorEnd->size() = 2 implies
    specializesFromLibrary("Links::binaryLinks")
```

checkConnectorObjectSpecialization

A `Connector` for an `AssociationStructure` must directly or indirectly specialize the base `Connector` *`Objects::linkObjects`* from the Kernel Semantic Library.

```
association->exists(oclIsKindOf(AssociationStructure)) implies
    specializesFromLibrary("Objects::linkObjects")
```

checkConnectorSpecialization

A `Connector` must directly or indirectly specialize the base `Connector` *`Links::links`* from the Kernel Semantic Library.

```
specializesFromLibrary("Links::links")
```

checkConnectorTypeFeaturing

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)
```

deriveConnectorRelatedFeature

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

```
relatedFeature = connectorEnd.ownedReferenceSubsetting->
    select(s | s <> null).subsettedFeature
```

deriveConnectorSourceFeature

The `sourceFeature` of a `Connector` is its first `relatedFeature` (if any).

```
sourceFeature =
    if relatedFeature->isEmpty() then null
    else relatedFeature->first()
    endif
```

deriveConnectorTargetFeature

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

```
targetFeature =
    if relatedFeature->size() < 2 then OrderedSet{}
    else
        relatedFeature->
            subSequence(2, relatedFeature->size())->
            asOrderedSet()
    endif
```

validateConnectorBinarySpecialization

If a `Connector` has more than two `connectorEnds`, then it must *not* specialize, directly or indirectly, the `Association` *`BinaryLink`* from the Kernel Semantic Library.

```
connectorEnds->size() > 2 implies
    not specializesFromLibrary("Links::BinaryLink")
```

validateConnectorRelatedFeatures

If a `Connector` is concrete (not abstract), then it must have at least two `relatedFeatures`.

```
not isAbstract implies relatedFeature->size() >= 2
```

8.3.4.5.4 Succession

Description

A `Succession` is a binary `Connector` that requires its `relatedFeatures` to happen separately in time.

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]

A `Step` that is typed by the `Behavior` *`TransitionPerformances::TransitionPerformance`* (from the Kernel Semantic 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

None.

Constraints

checkSuccessionSpecialization

A Succession must directly or indirectly specialize the Feature *Occurrences::happensBeforeLinks* from the Kernel Semantic Library.

```
specializesFromLibrary("Occurences::happensBeforeLinks")
```

deriveSuccessionEffectStep

The effectStep of a Succession is the fourth ownedFeature of its transitionStep, which must be a Step.

```
effectStep =
    if transitionStep = null or
       transitionStep.ownedFeature.size() < 4 or
       not transitionStep.ownedFeature->at(4).oclIsKindOf(Step)
    then Set{}
    else Set{transitionStep.ownedFeature->at(4).oclAsType(Step)}
    endif
```

deriveSuccessionGuardExpression

The guardExpression of a Succession is the third ownedFeature of its transitionStep, which must be an Expression.

```
guardExpression =
    if transitionStep = null or
       transitionStep.ownedFeature.size() < 3 or
       not transitionStep.ownedFeature->at(3).oclIsKindOf(Expression)
    then Set{}
    else Set{transitionStep.ownedFeature->at(3).oclAsType(Expression)}
    endif
```

deriveSuccessionTransitionStep

If the owningNamespace of a Succession is a Step that specializes *TransitionPerformances::TransitionPerformance*, then this is presumed to be the transitionStep for the Succession.

```
transitionStep =
    if owningNamespace.oclIsKindOf(Step) and
        owningNamespace.oclAsType(Step).
            specializesFromLibrary('TransitionPerformances::TransitionPerformance') then
        owningNamespace.oclAsType(Step)
    else null
    endif
```

deriveSuccessionTriggerStep

The triggerStep of a Succession is the second ownedFeature of its transitionStep, which must be a Step.

```
triggerStep =
    if transitionStep = null or
```

```
    transitionStep.ownedFeature.size() < 2 or
    not transitionStep.ownedFeature->at(2).oclIsKindOf(Step)
then Set{}
else Set{transitionStep.ownedFeature->at(2).oclAsType(Step)}
endif
```

8.3.4.6 Behaviors Abstract Syntax

8.3.4.6.1 Overview

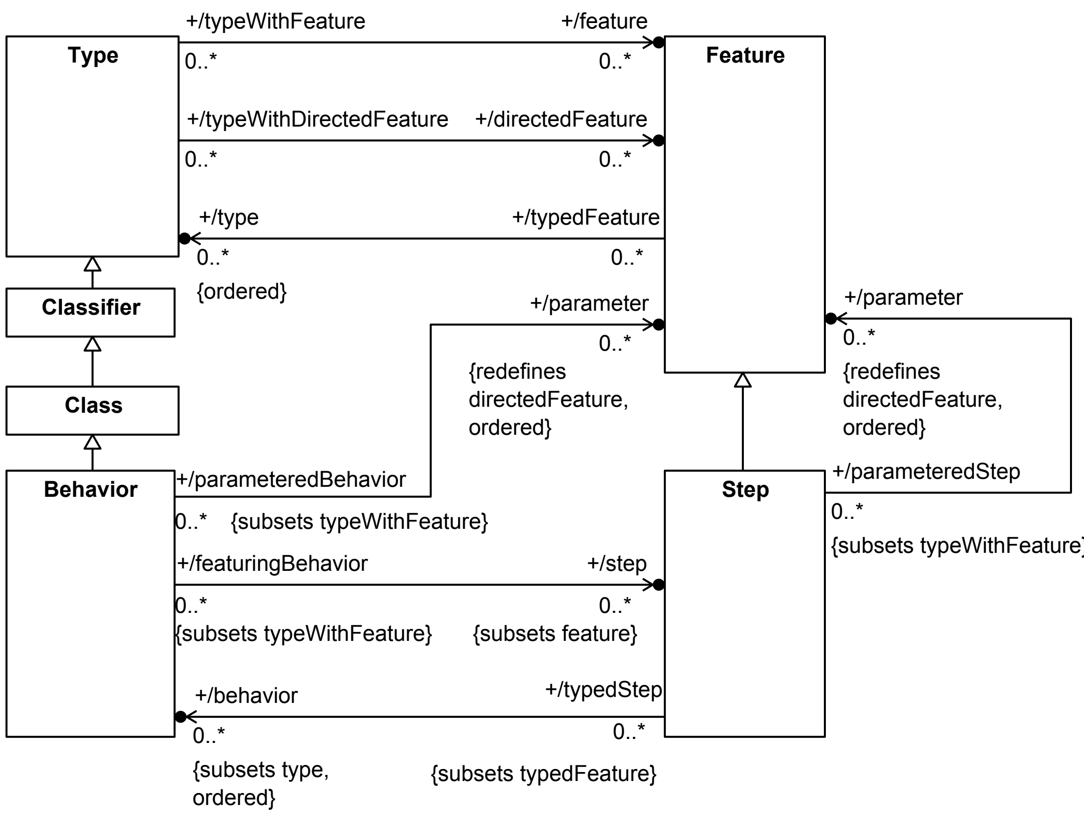

Figure 28. Behaviors

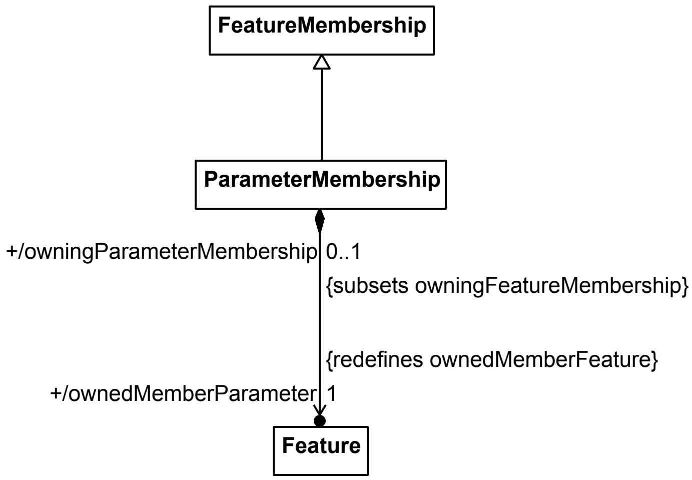

Figure 29. Parameter Memberships

8.3.4.6.2 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 defined as 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

None.

Constraints

checkBehaviorSpecialization

A `Behavior` must directly or indirectly specialize the base `Behavior` *`Performances::Performance`* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::Performance")
```

deriveBehaviorStep

The `steps` of a `Behavior` are its `features` that are `Steps`.

```
step = feature->selectByKind(Step)
```

8.3.4.6.3 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 a 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 [0..*] {subsets type, ordered}

The `Behaviors` that type this `Step`.

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

The `parameters` of this `Step`, which are defined as its `directedFeatures`, whose values are passed into and/or out of a performance of the `Step`.

Operations

None.

Constraints

checkStepEnclosedPerformanceSpecialization

A`Step` whose `owningType` is a `Behavior` or another `Step` must directly or indirectly specialize the `Step` *`Performances::Performance::enclosedPerformance`*.

```
owningType <> null and
    (owningType.oclIsKindOf(Behavior) or
     owningType.oclIsKindOf(Step)) implies
    specializesFromLibrary('Performances::Performance::enclosedPerformance')
```

checkStepOwnedPerformanceSpecialization

A composite `Step` whose `owningType` is a `Structure` or a `Feature` typed by a `Structure` must directly or indirectly specialize the `Step` *`Objects::Object::ownedPerformance`*.

```
isComposite and owningType <> null and
(owningType.oclIsKindOf(Structure) or
 owningType.oclIsKindOf(Feature) and
 owningType.oclAsType(Feature).type->
    exists(oclIsKindOf(Structure)) implies
    specializesFromLibrary('Objects::Object::ownedPerformance')
```

checkStepSpecialization

A Step must directly or indirectly specialize the base Step *Performances::performances* from the Kernel Semantic Library.

```
allSupertypes()->includes(resolveGlobal("Performances::performances"))
```

checkStepSubperformanceSpecialization

AStep whose owningType is a Behavior or another Step, and which is composite, must directly or indirectly specialize the Step *Performances::Performance::subperformance*.

```
owningType <> null and
    (owningType.oclIsKindOf(Behavior) or
     owningType.oclIsKindOf(Step)) and
    self.isComposite implies
    specializesFromLibrary('Performances::Performance::subperformance')
```

deriveStepBehavior

The behaviors of a Step are all its types that are Behaviors.

```
behavior = type->selectByKind(Behavior)
```

8.3.4.6.4 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.

Operations

None.

Constraints

validateParameterMembershipOwningType

A ParameterMembership must be owned by a Behavior or a Step.

```
owningType.oclIsKindOf(Behavior) or owningType.oclIsKindOf(Step)
```

validateParameterMembershipParameterHasDirection

The ownedMemberParameter of a ParameterMembership must have a non-null direction.

```
ownedMemberParameter.direction <> null
```

8.3.4.7 Functions Abstract Syntax

8.3.4.7.1 Overview

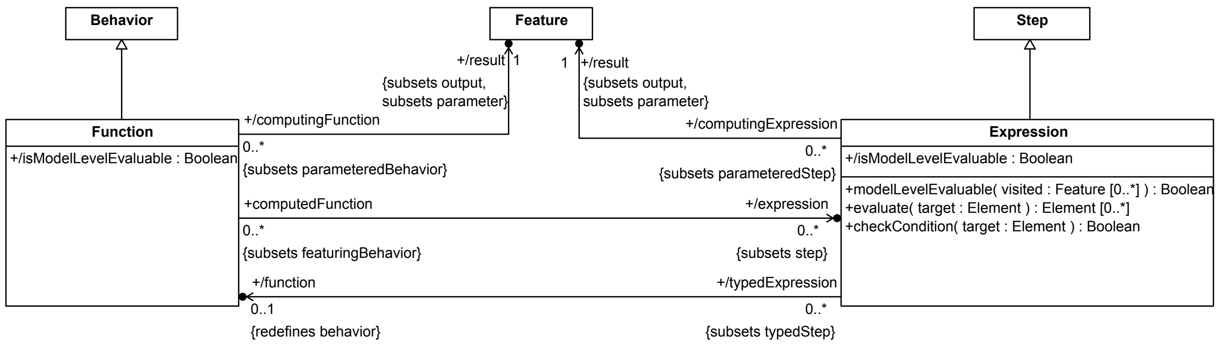

Figure 30. Functions

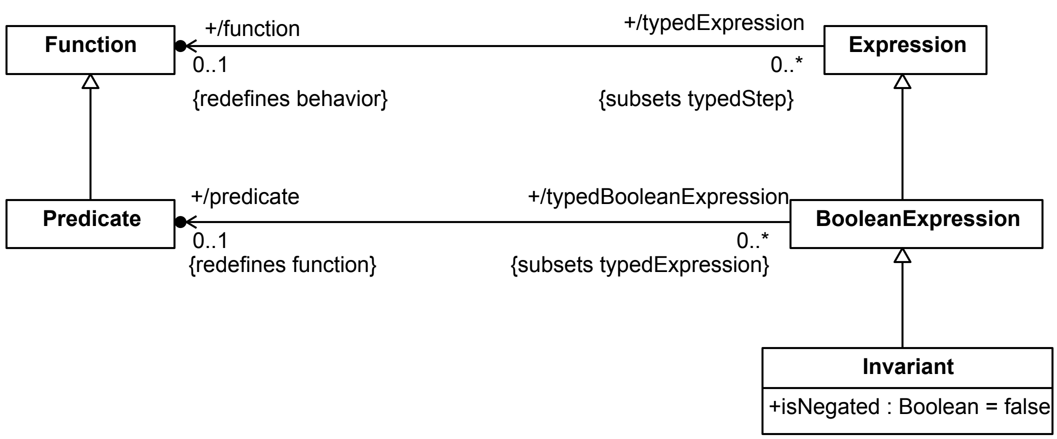

Figure 31. Predicates

Figure 32. Function Memberships

8.3.4.7.2 BooleanExpression

Description

A `BooleanExpression` is a *`Boolean`*-valued `Expression` whose type is a `Predicate`. It represents a logical condition resulting from the evaluation of the `Predicate`.

General Classes

Expression

Attributes

/predicate : Predicate [0..1] {redefines function}

The `Predicate` that types this `BooleanExpression`.

Operations

None.

Constraints

checkBooleanExpressionSpecialization

A BooleanExpression must directly or indirectly specialize the base BooleanExpression *`Performances::booleanEvaluations`* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::booleanEvaluations")
```

8.3.4.7.3 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 `Expression` in the tree.

General Classes

Step

Attributes

/function : Function [0..1] {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}

An `output` `parameter` of the `Expression` whose value is the result of the `Expression`. The result of an `Expression` is either inherited from its `function` or it is related to the `Expression` via a `ReturnParameterMembership`, in which case it redefines the `result` `parameter` of its `function`.

Operations

checkCondition(target : Element) : Boolean

Model-level evaluate this `Expression` with the given `target`. If the result is a `LiteralBoolean`, return its `value`. Otherwise return `false`.

```
body: let results: Sequence(Element) = evaluate(target) in
    result->size() = 1 and
    results->first().oclIsKindOf(LiteralBoolean) and
    results->first().oclAsType(LiteralBoolean).value
```

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`, which, for a fully evaluable `Expression`, will be a `LiteralExpression` or a `Feature` that is not an `Expression`.

```
pre: isModelLevelEvaluable
```

```
body: let resultExprs : Sequence(Expression) =
    ownedFeatureMembership->
        selectByKind(ResultExpressionMembership).
        ownedResultExpression in
if resultExpr->isEmpty() then Sequence{}
else resultExprs->first().evaluate(target)
endif
```

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

Return whether this `Expression` is model-level evaluable. The `visited` parameter is used to track possible circular `Feature` references. Such circular references are not allowed in model-level evaluable expressions.

An `Expression` that is not otherwise specialized is model-level evaluable if all of it has no `ownedSpecialziations` and all its (non-implicit) `features` are either `in` parameters, the `result` `parameter` or a result `Expression` owned via a `ResultExpressionMembership`. The `parameters` must not have any `ownedFeatures` or a `FeatureValue`, and the result `Expression` must be model-level evaluable.

```
body: ownedSpecialization->isEmpty() and
ownedFeature->forAll(f |
    (f.oclIsKindOf(Relationship) and
        f.oclAsType(Relationship).isImplicit) or
    (directionOf(f) = FeatureDirectionKind::_'in' or f = result) and
        f.ownedFeature->isEmpty() f.valuation = null and  or
    f.owningFeatureMembership.oclIsKindOf(ResultExpressionMembership) and
        f.oclAsType(Expression).modelLevelEvaluable(visited)
```

Constraints

checkExpressionResultBindingConnector

If an `Expression` has an `Expression` owned via a `ResultExpressionMembership`, then the owning `Expression` must also own a `BindingConnector` between its `result` `parameter` and the `result` `parameter` of the result `Expression`.

```
ownedMembership.selectByKind(ResultExpressionMembership)->
    forAll(mem | ownedFeature.selectByKind(BindingConnector)->
        exists(binding |
            binding.relatedFeature->includes(result) and
            binding.relatedFeature->includes(mem.ownedResultExpression.result)))
```

checkExpressionSpecialization

An Expression must directly or indirectly specialize the base Expression *`Performances::evaluations`* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::evaluations")
```

checkExpressionTypeFeaturing

If this `Expression` is owned by a `FeatureValue`, then it must have the same `featuringTypes` as the `featureWithValue` of the `FeatureValue`.

```
owningMembership <> null and
owningMembership.oclIsKindOf(FeatureValue) implies
    let featureWithValue : Feature =
        owningMembership.oclAsType(FeatureValue).featureWithValue in
    featuringType = featureWithValue.featuringType
```

deriveExpressionIsModelLevelEvaluable

Whether an Expression `isModelLevelEvaluable` is determined by the `modelLevelEvaluable()` operation.

```
isModelLevelEvaluable = modelLevelEvaluable(Set(Element){})
```

deriveExpressionResult

If an `Expression` has a `parameter` owned via a `ReturnParameterMembership`, then that is its `result` `parameter`. Otherwise, its `result` `parameter` is the `result` `parameter` inherited from its `function`.

```
result =
    let resultParams : Sequence(Feature) =
        ownedFeatureMemberships->
            selectByKind(ReturnParameterMembership).
            ownedParameterMember in
    if resultParams->notEmpty() then resultParams->first()
    else if function <> null then function.result
    else null
    endif endif
```

validateExpressionResultParameterMembership

An `Expression` must own at most one `ResultParameterMembership`.

```
ownedFeatureMembership->
    selectByKind(ReturnParameterMembership)->
    size() <= 1
```

8.3.4.7.4 Function

Description

A Function is a Behavior that has an out parameter that is identified as its result. A Function represents the performance of a calculation that produces the values of its result parameter. This calculation may be decomposed into Expressionssteps of the Function.

General Classes

Behavior

Attributes

/expression : Expression [0..*] {subsets 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. Certain Functions from the Kernel Functions Library are considered to have isModelLevelEvaluable = true. For all other Functions it is false.

Note: See the specification of the KerML concrete syntax notation for Expressions for an identification of which library Functions are model-level evaluable.

/result : Feature {subsets parameter, output}

The result parameter of the Function, which is owned by the Function via a ReturnParameterMembership.

Operations

None.

Constraints

checkFunctionResultBindingConnector

If a Function has an Expression owned via a ResultExpressionMembership, then the owning Function must also own a BindingConnector between its result parameter and the result parameter of the result Expression.

```
ownedMembership.selectByKind(ResultExpressionMembership)->
    forAll(mem | ownedFeature.selectByKind(BindingConnector)->
        exists(binding |
            binding.relatedFeature->includes(result) and
            binding.relatedFeature->includes(mem.ownedResultExpression.result)))
```

checkFunctionSpecialization

A Function must directly or indirectly specialize the base Function *Performances::Evaluation* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::Evaluation")
```

deriveFunctionResult

The `result` `parameter` of a `Function` is its `parameter` owned via a `ReturnParameterMembership` (if any).

```
result =
    let resultParams : Sequence(Feature) =
        ownedFeatureMemberships->
            selectByKind(ReturnParameterMembership).
            ownedParameterMember in
    if resultParams->notEmpty() then resultParams->first()
    else null
    endif
```

validateFunctionResultParameterMembership

A `Function` must own at most one `ResultParameterMembership`.

```
ownedFeatureMembership->
    selectByKind(ReturnParameterMembership)->
    size() <= 1
```

8.3.4.7.5 Invariant

Description

An `Invariant` is a `BooleanExpression` that is asserted to have a specific *Boolean* result value. If `isNegated = false`, then the result is asserted to be true. If `isNegated = true`, then 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

None.

Constraints

checkInvariantSpecialization

An Invariant must directly or indirectly specialize either of the following BooleanExpressions from the Kernel Semantic Library: *Performances::trueEvaluations*, if `isNegated = false`, or *Performances::falseEvaluations*, if `isNegated = true`.

```
if isNegated then
    specializesFromLibrary("Performances::falseEvaluations")
else
    specializesFromLibrary("Performances::trueEvaluations")
endif
```

8.3.4.7.6 Predicate

Description

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

General Classes

Function

Attributes

None.

Operations

None.

Constraints

checkPredicateSpecialization

A Predicate must directly or indirectly specialize the base Predicate *`Performances::BooleanEvaluation`* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::BooleanEvaluation")
```

8.3.4.7.7 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 owning `Function` or `Expression`.

General Classes

FeatureMembership

Attributes

/ownedResultExpression : Expression {redefines ownedMemberFeature}

The `Expression` that provides the result for the owner of the `ResultExpressionMembership`.

Operations

None.

Constraints

validateResultExpressionMembershipOwningType

The `owningType` of a `ResultExpressionMembership` must be a `Function` or `Expression`.

```
owningType.oclIsKindOf(Function) or owningType.oclIsKindOf(Expression)
```

8.3.4.7.8 ReturnParameterMembership

Description

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

General Classes

ParameterMembership

Attributes

None.

Operations

None.

Constraints

validateReturnParameterMembershipOwningType

The `owningType` of a `ReturnParameterMembership` must be a `Function` or `Expression`.

```
owningType.oclIsKindOf(Function) or owningType.oclIsKindOf(Expression)
```

validateReturnParameterMembershipParameterHasDirectionOut

The `ownedMemberParameter` of a `ReturnParameterMembership` must have a `direction = out`.

```
ownedMemberParameter.direction = ParameterDirectionKind::out
```

8.3.4.8 Expressions Abstract Syntax

8.3.4.8.1 Overview

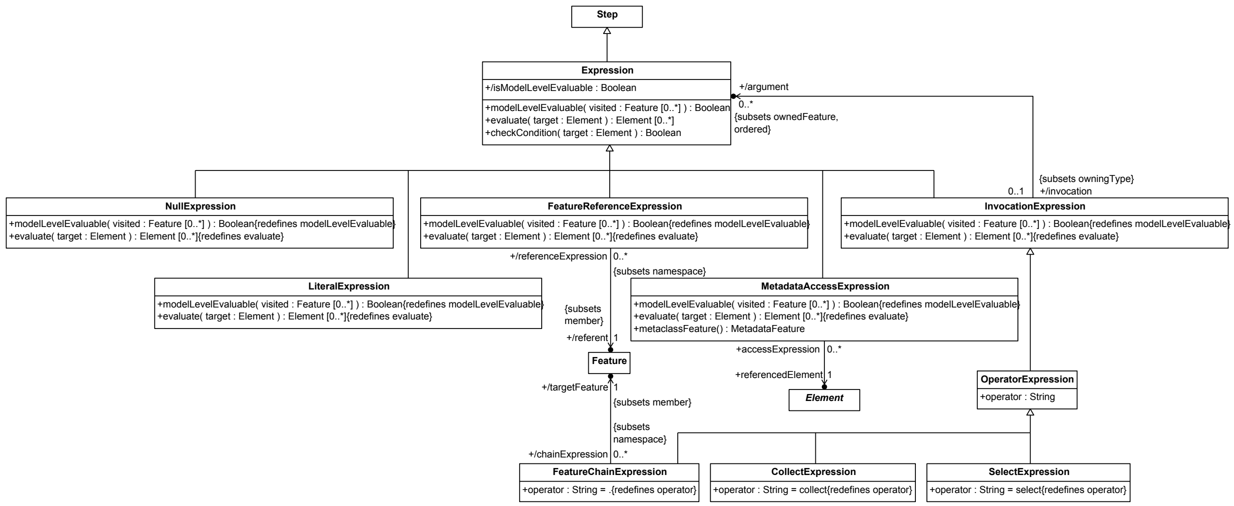

Figure 33. Expressions

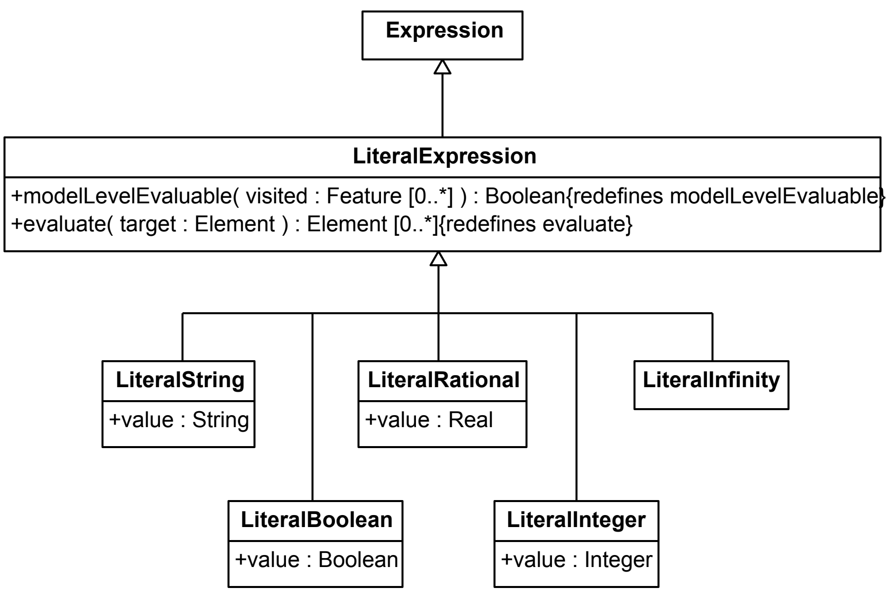

Figure 34. Literal Expressions

8.3.4.8.2 CollectExpression

Description

A `CollectExpression` is an `OperatorExpression` whose `operator` is `"collect"`, which resolves to the `Function` *`ControlFunctions::collect`* from the Kernel Functions Library.

General Classes

OperatorExpression

Attributes

operator : String {redefines operator}

Operations

None.

Constraints

validateCollectExpressionOperator

The `operator` of a `CollectExpression` must be `"collect"`.

```
operator = "collect"
```

8.3.4.8.3 FeatureChainExpression

Description

A `FeatureChainExpression` is an `OperatorExpression` whose operator is `"."`, which resolves to the `Function` *`ControlFunctions::'.'`* from the Kernel Functions Library. It evaluates to the result of chaining the `result` `Feature` of its single `argument` `Expression` with its `targetFeature`.

General Classes

OperatorExpression

Attributes

operator : String {redefines operator}

/targetFeature : Feature {subsets member}

The `Feature` that is accessed by this `FeatureChainExpression, which is its first non-parameter member.`

Operations

sourceTargetFeature() : Feature [0..1]

Return the first `ownedFeature` of the first owned input `parameter` of this `FeatureChainExpression` (if any).

```
body: let inputParameters : Feature = ownedFeatures->
    select(direction = _'in') in
if inputParameters->isEmpty() or
   inputParameters->first().ownedFeature->isEmpty()
then null
else inputParameters->first().ownedFeature->first()
endif
```

Constraints

checkFeatureChainExpressionSourceTargetRedefinition

The first `ownedFeature` of the first owned input `parameter` of a `FeatureChainExpression` must redefine its `targetFeature`.

```
let sourceParameter : Feature = sourceTargetFeature() in
sourceTargetFeature <> null and
sourceTargetFeature.redefines(targetFeature)
```

checkFeatureChainExpressionTargetRedefinition

The first `ownedFeature` of the first owned input `parameter` of a `FeatureChainExpression` must redefine the `Feature` *`ControlFunctions::'.'::source::target`* from the Kernel Functions Library.

```
let sourceParameter : Feature = sourceTargetFeature() in
sourceTargetFeature <> null and
sourceTargetFeature.redefinesFromLibrary("ControlFunctions::'.'::source::target")
```

deriveFeatureChainExpressionTargetFeature

The `targetFeature` of a `FeatureChainExpression` is the `memberElement` of its first `ownedMembership` that is not a `ParameterMembership`.

```
targetFeature =
    let nonParameterMemberships : Sequence(Membership) = ownedMembership->
        reject(oclIsKindOf(ParameterMembership)) in
    if nonParameterMemberships->isEmpty() or
       not nonParameterMemberships->first().memberElement.oclIsKindOf(Feature)
    then null
    else nonParameterMemberships->first().memberElement.oclAsType(Feature)
    endif
```

8.3.4.8.4 FeatureReferenceExpression

Description

A `FeatureReferenceExpression` is an `Expression` whose `result` is bound to a `referent Feature`.

General Classes

Expression

Attributes

/referent : Feature {subsets member}

The `Feature` that is referenced by this `FeatureReferenceExpression`, which is its first non-`parameter member`.

Operations

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

First, determine a `value Expression` for the `referent`:

- If the `target Element` is a Type that has a `feature` that is the `referent` or (directly or indirectly) redefines it, then the `value Expression` of the `FeatureValue` for that `feature` (if any).
- Else, if the `referent` has no `featuringTypes`, the `value Expression` of the `FeatureValue` for the `referent` (if any).

Then:

- If such a value `Expression` exists, return the result of evaluating that `Expression` on the `target`.
- Else, if the `referent` is not an `Expression`, return the `referent`.
- Else return the 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
```

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

A `FeatureReferenceExpression` is model-level evaluable if it's `referent`

- conforms to the self-reference feature *`Anything::self`*;
- is an `Expression` that is model-level evaluable;
- has an `owningType` that is a `Metaclass` or `MetadataFeature`; or
- has no `featuringTypes` and, if it has a `FeatureValue`, the `value Expression` is model-level evaluable.

```
body: referent.conformsTo("Anything::self") or
visited->excludes(referent) and
(referent.oclIsKindOf(Expression) and
    referent.oclAsType(Expression).modelLevelEvaluable(visited->including(referent)) or
referent.owningType <> null and
    (referent.owningType.isOclKindOf(MetaClass) or
    referent.owningType.isOclKindOf(MetadataFeature)) or
referent.featuringType->isEmpty() and
    (referent.valuation = null or
    referent.valuation.modelLevelEvaluable(visited->including(referent))))
```

Constraints

checkFeatureReferenceExpressionBindingConnector

A `FeatureReferenceExpression` must have an `ownedMember` that is a `BindingConnector` between the `referent` and `result` of the `FeatureReferenceExpression`.

```
ownedMember->selectByKind(BindingConnector)->exists(b |
    b.relatedFeatures->includes(targetFeature) and
    b.relatedFeatures->includes(result))
```

deriveFeatureReferenceExpressionReferent

The `targetFeature` of a `FeatureChainExpression` is the `memberElement` of its first `ownedMembership` that is not a `ParameterMembership`.

```
referent =
    let nonParameterMemberships : Sequence(Membership) = ownedMembership->
        reject(oclIsKindOf(ParameterMembership)) in
    if nonParameterMemberships->isEmpty() or
       not nonParameterMemberships->first().memberElement.oclIsKindOf(Feature)
    then null
    else nonParameterMemberships->first().memberElement.oclAsType(Feature)
    endif
```

8.3.4.8.5 InvocationExpression

Description

An `InvocationExpression` is an `Expression` each of whose input `parameters` are bound to the `result` of an `argument` Expression.

General Classes

Expression

Attributes

/argument : Expression [0..*] {subsets ownedFeature, ordered}

The `value` `Expressions` of the `FeatureValues` of the owned input `parameters` 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.

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

An `InvocationExpression` is model-level evaluable if all its `argument` `Expressions` are model-level evaluable and its `function` is model-level evaluable.

```
body: argument->forAll(modelLevelEvaluable(visited)) and
    function.isModelLevelEvaluable
```

Constraints

checkInvocationExpressionConstructorBindingConnector

If an `InvocationExpression` does not have an `ownedTyping` that is a `Behavior` or an `ownedSubsetting` of a `Feature` that is typed by a `Behavior`, then it must own a `BindingConnector between itself and its result parameter.`

```
not ownedTyping->exists(oclIsKindOf(Behavior)) and
not ownedSubsetting.subsettedFeature.type->exists(oclIsKindOf(Behavior)) implies
    ownedFeature.selectByKind(BindingConnector)->exists(
        relatedFeature->includes(self) and
```

```
        relatedFeature->includes(result))
```

checkInvocationExpressionDefaultValueBindingConnector

An `InvocationExpression` must own a `BindingConnector` between the `featureWithValue` and `value` `Expression` of any `FeatureValue` that is the effective default value for a `Feature` of the invoked `Type` of the `InvocationExpression`.

```
TBD
```

deriveInvocationExpressionArgument

The `arguments` of an `InvocationExpression` are the `value` `Expressions` of the `FeatureValues` of its owned input `parameters`.

```
ownedFeature->
    select(direction = _'in').valuation->
    select(v | v <> null).value
```

8.3.4.8.6 LiteralBoolean

Description

`LiteralBoolean` is a `LiteralExpression` that provides a *`Boolean`* value as a result. Its `result` `parameter` must have type *`Boolean`*.

General Classes

LiteralExpression

Attributes

value : Boolean

The *`Boolean`* value that is the result of evaluating this `LiteralBoolean`.

Operations

None.

Constraints

None.

8.3.4.8.7 LiteralExpression

Description

A `LiteralExpression` is an `Expression` that provides a basic *`DataValue`* as a result.

General Classes

Expression

Attributes

None.

Operations

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

The model-level value of a `LiteralExpression` is itself.

body: `Sequence{self}`

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

A `LiteralExpression` is always model-level evaluable.

body: `true`

Constraints

checkLiteralExpressionSpecialization

A `LiteralExpression` must directly or indirectly specialize the base `LiteralExpression` *`Performances::literalEvaluations`* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::literalEvaluations")
```

deriveLiteralExpressionIsModelLevelEvaluable

A `LiteralExpression` is always model-level evaluable.

```
isModelLevelEvaluable = true
```

8.3.4.8.8 LiteralInfinity

Description

A `LiteralInfinity` is a `LiteralExpression` that provides the positive infinity value (`*`). It's `result` must have the type *`Positive`*.

General Classes

LiteralExpression

Attributes

None.

Operations

None.

Constraints

None.

8.3.4.8.9 LiteralInteger

Description

A `LiteralInteger` is a `LiteralExpression` that provides an *`Integer`* value as a result. Its `result parameter` must have the type *`Integer`*.

General Classes

LiteralExpression

Attributes

value : Integer

The *`Integer`* value that is the result of evaluating this `LiteralInteger`.

Operations

None.

Constraints

None.

8.3.4.8.10 LiteralRational

Description

A `LiteralRational` is a `LiteralExpression` that provides a *`Rational`* value as a result. Its `result parameter` must have the type *`Rational`*.

General Classes

LiteralExpression

Attributes

value : Real

The value whose rational approximation is the result of evaluating this `LiteralRational`.

Operations

None.

Constraints

None.

8.3.4.8.11 LiteralString

Description

A `LiteralString` is a `LiteralExpression` that provides a *`String`* value as a result. Its `result` `parameter` must have the type *`String`*.

General Classes

LiteralExpression

Attributes

value : String

The *`String`* value that is the result of evaluating this `LiteralString`.

Operations

None.

Constraints

None.

8.3.4.8.12 MetadataAccessExpression

Description

A `MetadataAccessExpression` is an `Expression` whose `result` is a sequence of instances of `Metaclasses` representing all the `MetadataFeature` annotations of the `referencedElement`. In addition, the sequence includes an instance of the reflective `Metaclass` corresponding to the MOF class of the `referencedElement`, with values for all the abstract syntax properties of the `referencedElement`.

General Classes

Expression

Attributes

referencedElement : Element

The `Element` whose metadata is being accessed.

Operations

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

Return the `ownedElements` of the `referencedElement` that are `MetadataFeatures` and have the `referencedElement` as an `annotatedElement`, plus a `MetadataFeature` whose `annotatedElement` is the `referencedElement`, whose `metaclass` is the reflective `Metaclass` corresponding to the MOF class of the `referencedElement` and whose `ownedFeatures` are bound to the values of the MOF properties of the `referencedElement`.

```
body: referencedElement.ownedElement->
    select(oclIsKindOf(MetadataFeature)
        and annotatedElement->includes(referencedElement))->
    including(metaclassFeature())
```

metaclassFeature() : MetadataFeature

Return a `MetadataFeature` whose `annotatedElement` is the `referencedElement`, whose `metaclass` is the reflective `Metaclass` corresponding to the MOF class of the `referencedElement` and whose `ownedFeatures` are bound to the MOF properties of the `referencedElement`.

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

A `MetadataAccessExpression` is always model-level evaluable.

body: `true`

Constraints

checkMetadataAccessExpressionSpecialization

A `MetadataAccessExpression` must directly or indirectly specialize the base `MetadataAccessExpression` *`Performances::metadataAccessEvaluations`* from the Kernel Semantic Library.

`specializesFromLibrary("Performances::metadataAccessEvaluations")`

8.3.4.8.13 NullExpression

Description

A `NullExpression` is an `Expression` that results in a null 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{}`

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

A `NullExpression` is always model-level evaluable.

body: `true`

Constraints

checkNullExpressionSpecialization

A `NullExpression` must directly or indirectly specialize the base `NullExpression` *`Performances::nullEvaluations`* from the Kernel Semantic Library.

```
specializesFromLibrary("Performances::nullEvaluations")
```

8.3.4.8.14 OperatorExpression

Description

An `OperatorExpression` is an `InvocationExpression` whose `function` is determined by resolving its `operator` in the context of one of the standard packages from the Kernel Function Library.

General Classes

InvocationExpression

Attributes

operator : String

An `operator` symbol that names a corresponding `Function` from one of the standard packages from the Kernel Function Library .

Operations

None.

Constraints

checkOperatorExpressionSpecialization

The `function` of an `OperatorExpression` must be the resolution of the `operator` from one of the packages *`BaseFunctions`*, *`DataFunctions`*, or *`ControlFunctions`* from the Kernel Function Library.

```
let libFunctions : Sequence(Element) =
    Sequence{"BaseFunctions", "DataFunctions", "ControlFunctions"}->
    collect(ns | resolveGlobal(ns + "::'" + operator + "'")) in
libFunctions->includes(function)
```

8.3.4.8.15 SelectExpression

Description

A `SelectExpression` is an `OperatorExpression` whose operator is `"select"`, which resolves to the `Function` *`ControlFunctions::select`* from the Kernel Functions Library.

General Classes

OperatorExpression

Attributes

operator : String {redefines operator}

Operations

None.

Constraints

None.

8.3.4.9 Interactions Abstract Syntax

8.3.4.9.1 Overview

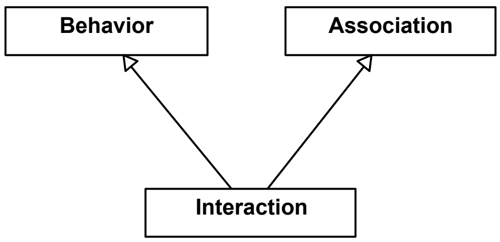

Figure 35. Interactions

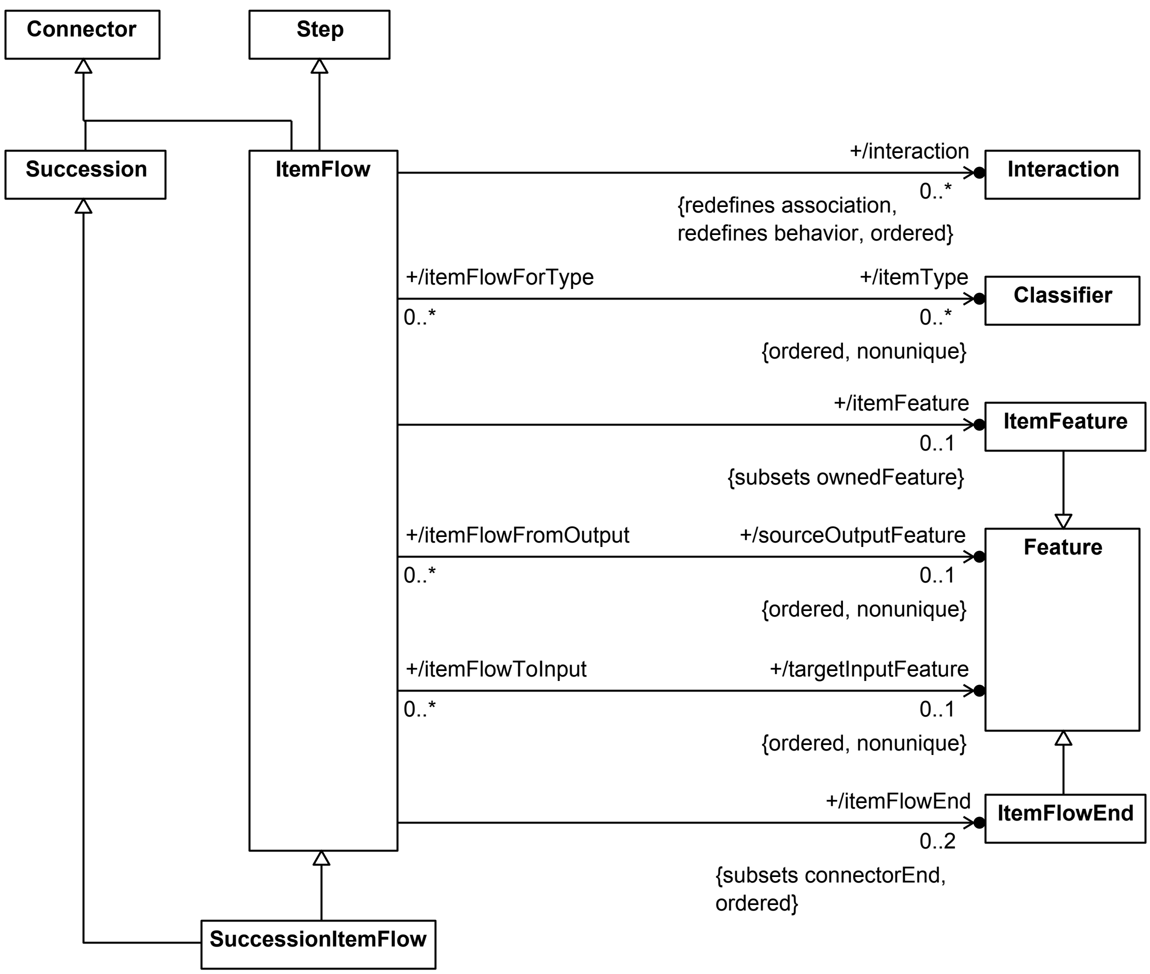

Figure 36. Item Flows

8.3.4.9.2 ItemFeature

Description

An `ItemFeature` is the `ownedFeature` of an `ItemFlow` that identifies the things carried by the kinds of transfers that are instances of the `ItemFlow`.

General Classes

Feature

Attributes

None.

Operations

None.

Constraints

checkItemFeatureRedefinition

An `ItemFeature` must redefine the `Feature` *`Transfers::Transfer::item`* from the Kernel Semantic Library.

```
ownedRedefinition.redefinedFeature->
    redefinesFromLibrary("Transfers::Transfer::item")
```

8.3.4.9.3 ItemFlow

Description

An `ItemFlow` is a `Step` that represents the transfer of objects or data values from one `Feature` to another. `ItemFlows` can take non-zero time to complete.

General Classes

Connector
Step

Attributes

/interaction : Interaction [0..*] {redefines association, behavior, ordered}

The `Interactions` that type this `ItemFlow`. `Interactions` are both `Associations` and `Behaviors`, which can type `Connectors` and `Steps`, respectively.

/itemFeature : ItemFeature [0..1] {subsets ownedFeature}

The `ownedFeature` of the `ItemFlow` that is an `ItemFeature` (if any).

/itemFlowEnd : ItemFlowEnd [0..2] {subsets connectorEnd, ordered}

The `connectorEnds` of this `ItemFlow` that are `ItemFlowEnds`.

/itemType : Classifier [0..*] {ordered, nonunique}

The type of values transferred, which is the `type` of the `itemFeature` of the `ItemFlow`.

/sourceOutputFeature : Feature [0..1] {ordered, nonunique}

The `Feature` that provides the items carried by the `ItemFlow`. It must be an owned `output` of the `source` of the `ItemFlow`.

/targetInputFeature : Feature [0..1] {ordered, nonunique}

The `Feature` that receives the values carried by the `ItemFlow`. It must be an owned `output` of the target participant of the `ItemFlow`.

Operations

None.

Constraints

checkItemFlowSpecialization

An `ItemFlow` must directly or indirectly specialize the `Step` *`Transfers::transfers`* from the Kernel Semantic Library. In addition, if the `ItemFlow` has `ItemFlowEnds`, it must specialize *`Transfers::flowTransfers`*.

```
if itemFlowEnds->isEmpty() then
    specializesFromLibrary("Transfers::transfers")
else
    specializesFromLibrary("Transfers::flowTransfers")
endif
```

deriveItemFlowItemFeature

The `itemFeature` of an `ItemFlow` is the single one of its `ownedFeatures` that is an `ItemFeature`.

```
itemFeature =
    let itemFeatures : Sequence(ItemFeature) =
        ownedFeature->selectByKind(ItemFeature) in
    if itemFeatures->isEmpty() then null
    else itemFeatures->first()
    endif
```

deriveItemFlowItemFlowEnd

The `itemFlowEnds` of a `ItemFlow` are all its `connectorEnds` that are `ItemFlowEnds`.

```
itemFlowEnd = connectorEnd->selectByKind(ItemFlowEnd)
```

deriveItemFlowItemType

The `itemTypes` of an `ItemFlow` are the `types` of the `itemFeature` of the `ItemFlow` (if any).

```
itemType =
    if itemFeature = null then Sequence{}
    else itemFeature.type
    endif
```

deriveItemFlowSourceOutputFeature

The `sourceOutputFeature` of a `ItemFlow` is the first `ownedFeature` of the first `connectorEnd` of the `ItemFlow`.

```
sourceOutputFeature =
    if connectorEnd->isEmpty() or
        connectorEnd.ownedFeature->isEmpty()
    then null
    else connectorEnd.ownedFeature->first()
    endif
```

deriveItemFlowTargetInputFeature

The `targetInputFeature` of a `ItemFlow` is the first `ownedFeature` of the second `connectorEnd` of the `ItemFlow`.

```
targetInputFeature =
    if connectorEnd->size() < 2 or
        connectorEnd->at(2).ownedFeature->isEmpty()
    then null
    else connectorEnd->at(2).ownedFeature->first()
    endif
```

validateItemFlowItemFeature

An `ItemFlow` must have at most one `ownedFeature` that is an `ItemFlow`.

```
ownedFeature->selectByKind(ItemFeature)->size() <= 1
```

8.3.4.9.4 ItemFlowEnd

Description

An `ItemFlowEnd` is a `Feature` that is one of the `connectorEnds` giving the *`source`* or *`target`* of an `ItemFlow`. For `ItemFlows` typed by *`FlowTransfer`* or its specializations, `ItemFlowEnds` must have exactly one `ownedFeature`, which redefines *`Transfer::source::sourceOutput`* or *`Transfer::target::targetInput`* and redefines the corresponding feature of the `relatedElement` for its end.

General Classes

Feature

Attributes

None.

Operations

None.

Constraints

validateItemFlowEndIsEnd

An `ItemFlowEnd` must be an end `Feature`.

isEnd

validateItemFlowEndNestedFeature

An ItemFlowEnd must have exactly one ownedFeature.

```
ownedFeature->size() = 1
```

validateItemFlowEndOwningType

The owningType of an ItemFlow must be an ItemFlow.

```
owningType <> null and owningType.oclIsKindOf(ItemFlow)
```

8.3.4.9.5 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

None.

Constraints

None.

8.3.4.9.6 SuccessionItemFlow

Description

A SuccessionItemFlow is an ItemFlow that also provides temporal ordering. It classifies *Transfers* that cannot start until the source *Occurrence* has completed and that must complete before the target *Occurrence* can start.

General Classes

Succession
ItemFlow

Attributes

None.

Operations

None.

Constraints

checkSuccessionItemFlowSpecialization

A `SuccessionItemFlow` must directly or indirectly specialize the `Step` *`Transfers::flowTransfersBefore`* from the Kernel Semantic Library.

```
specializesFromLibrary("Transfers::flowTransfersBefore")
```

8.3.4.10 Feature Values Abstract Syntax

8.3.4.10.1 Overview

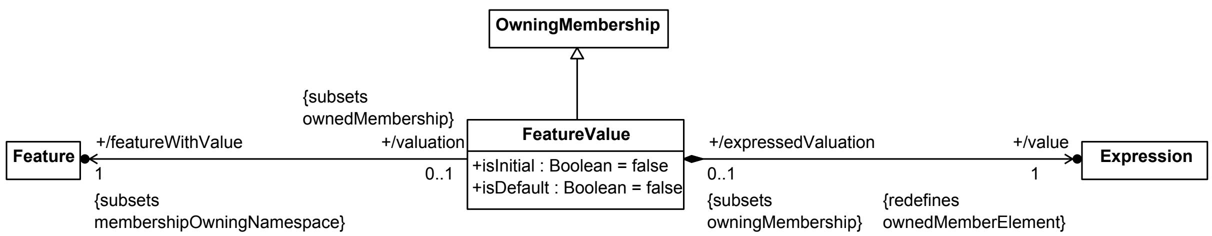

Figure 37. Feature Values

8.3.4.10.2 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`.

The result of the `value Expression` is bound to the `featureWithValue` using a `BindingConnector`. If `isInitial = false`, then the `featuringType` of the `BindingConnector` is the same as the `featuringType` of the `featureWithValue`. If `isInitial = true`, then the `featuringType` of the `BindingConnector` is restricted to its `startShot`.

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 or 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

None.

Constraints

checkFeatureValueBindingConnector

If `isDefault = false`, then the `featureWithValue` must have an `ownedMember` that is a `BindingConnector` whose `relatedElements` are the `featureWithValue` and the `result` of the `value` `Expression`. If `isInitial = false`, then this `BindingConnector` must have `featuringTypes` that are the same as those of the `featureWithValue`. If `isInitial = true`, then the `BindingConnector` must have *`that.startShot`* `as its featuringType`.

```
not isDefault implies
    featureWithValue.ownedMember->
        selectByKind(BindingConnector)->exists(b |
            b.relatedFeature->includes(featureWithValue) and
            b.relatedFeature->includes(value.result) and
            if not isInitial then
                b.featuringType = featureWithValue.featuringType
            else
                b.featuringType->exists(t |
                    t.oclIsKindOf(Feature) and
                    t.oclAsType(Feature).chainingFeature =
                        Sequence{
                            resolveGlobal("Base::things::that"),
                            resolveGlobal("Occurrences::Occurrence::startShot")
                        }
                )
            endif)
```

8.3.4.11 Multiplicities Abstract Syntax

8.3.4.11.1 Overview

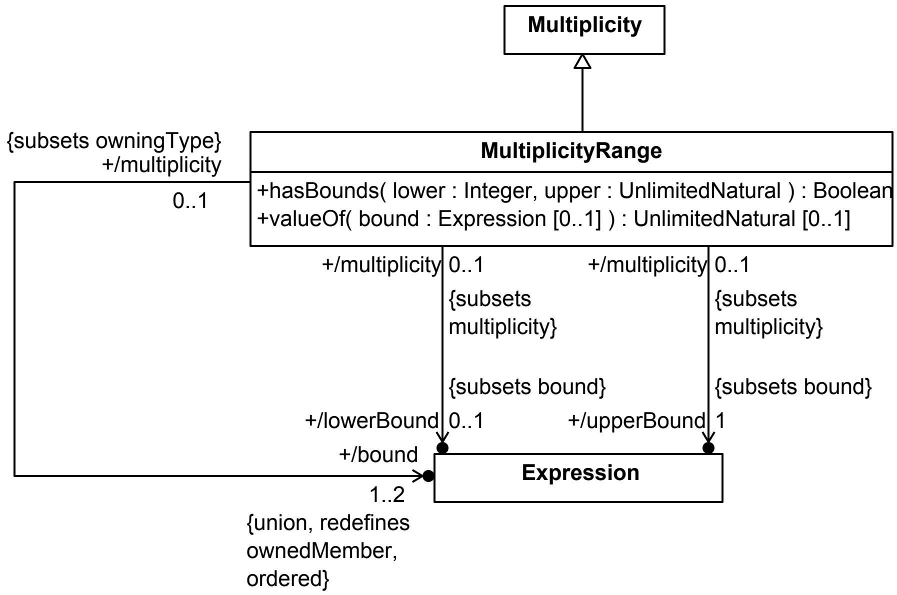

Figure 38. Multiplicities

8.3.4.11.2 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 these `Expressions` shall be of type *`Natural`*. 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. If no `lowerBound` `Expression`, then the default is that the lower bound has the same value as the upper bound, except if the `upperBound` evaluates to `*`, in which case the default for the lower bound is 0.

General Classes

Multiplicity

Attributes

/bound : Expression [1..2] {redefines ownedMember, ordered, union}

The owned `Expressions` of the `MultiplicityRange` whose results provide its bounds. These must be the only `ownedMembers` of the `MultiplicityRange`.

/lowerBound : Expression [0..1] {subsets bound}

The `Expression` whose result provides the lower bound of the `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

hasBounds(lower : Integer, upper : UnlimitedNatural) : Boolean

Check whether this `MultiplicityRange` represents the range bounded by the given values `lower` and `upper`, presuming the `lowerBound` and `upperBound` `Expressions` are model-level evaluable.

```
body: valueOf(upperBound) = upper and
let lowerValue: UnlimitedNatural = valueOf(lowerBound) in
(lowerValue = lower or
 lowerValue = null and
    (lower = upper or
     lower = 0 and upper = *))
```

valueOf(bound : Expression [0..1]) : UnlimitedNatural [0..1]

Evaluate the given `bound` `Expression` (at model level) and return the result represented as a MOF `UnlimitedNatural` value.

```
body: if bound = null or not bound.isModelLevelEvaluable then
    null
else
    let boundEval: Sequence(Element) = bound.evaluate(owningType) in
    if boundEval->size() <> 1 then null else
        let valueEval: Element = boundEval->at(1) in
        if valueEval.oclIsKindOf(LiteralInfinity) then *
        else if valueEval.oclIsKindOf(LiteralInteger) then
            let value : Integer =
                valueEval.oclAsKindOf(LiteralInteger).value in
            if value >= 0 then value else null endif
        else null
        endif endif
    endif
endif
```

Constraints

checkMultiplicityRangeExpressionTypeFeaturing

The `bounds` of a `MultiplicityRange must have the same featuringTypes as the MultiplicityRange.`

```
bound->forAll(b | b.featuringType = self.featuringType)
```

deriveMultiplicityRangeLowerBound

If a `MultiplicityRange` has two `ownedMembers`, then the `lowerBound` is the first `ownedMember`, otherwise it is `null`.

```
lowerBound =
    let ownedMembers : Sequence(Element) =
        ownedMembership->selectByKind(OwningMembership).ownedMember in
    if ownedMembers->size() < 2 or
        not ownedMembers->first().oclIsKindOf(Expression) then null
```

```
    else ownedMembers->first().oclAsType(Expression)
    endif
```

deriveMultiplicityRangeUpperBound

If a `MultiplicityRange` has `ownedMembers` when considered as a `Feature`, then `upperBound` is the last of those, which must be an `Expression`.

```
upperBound =
    let ownedMembers : Sequence(Element) =
        ownedMembership->selectByKind(OwningMembership).ownedMember in
    if ownedMembers->isEmpty() or
       not ownedMembers->last().oclIsKindOf(Expression)
    then null
    else ownedMembers->last().oclAsType(Expression)
    endif
```

validateMultiplicityRangeBoundResultTypes

The `results` of the `bound Expression(s)` of a `MultiplicityRange` must be typed by *`ScalarValues::Natural`* from the Kernel Data Types Library.

```
bound.result->forAll(specializesFromLibrary('ScalarValues::Natural'))
```

8.3.4.12 Metadata Abstract Syntax

8.3.4.12.1 Overview

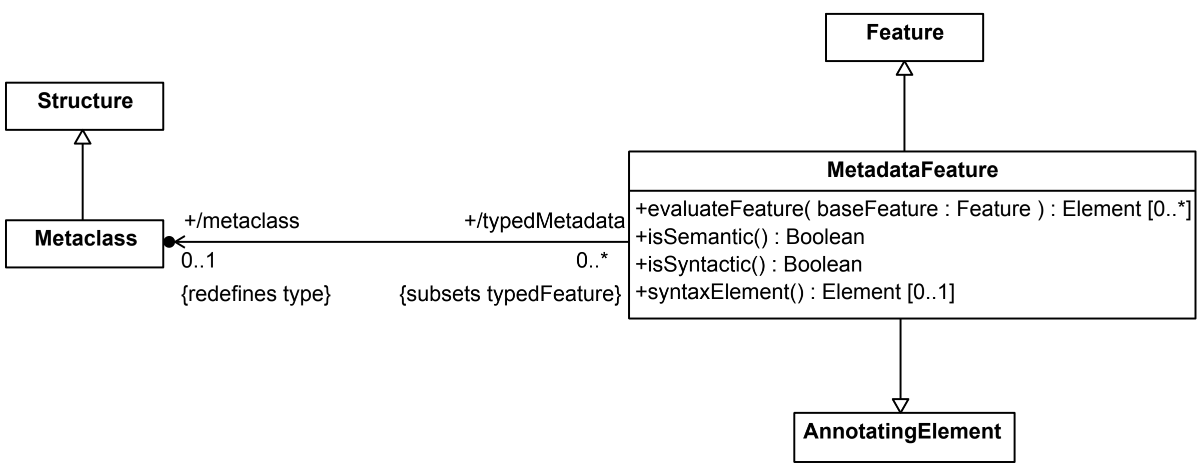

Figure 39. Metadata Annotation

8.3.4.12.2 Metaclass

Description

A `Metaclass` is a `Structure` used to type `MetadataFeatures`.

General Classes

Structure

Attributes

None.

Operations

None.

Constraints

checkMetaclassSpecialization

A `Metaclass` must directly or indirectly specialize the base `Metaclass` *`Metaobjects::Metaobject`* from the Kernel Semantic Library.

```
specializesFromLibrary("Metaobjects::Metaobject")
```

8.3.4.12.3 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.

General Classes

AnnotatingElement
Feature

Attributes

/metaclass : Metaclass [0..1] {redefines type}

The `type` of this `MetadataFeature`, which must be a `Metaclass`.

Operations

evaluateFeature(baseFeature : Feature) : Element [0..*]

If the given `baseFeature` is a `feature` of this `MetadataFeature`, or is directly or indirectly redefined by a `feature`, then return the result of evaluating the appropriate (model-level evaluable) `value` `Expression` for it (if any), with the MetadataFeature as the target.

```
body: let selectedFeatures : Sequence(Feature) = feature->
    select(closure(ownedRedefinition.redefinedFeature)->
           includes(baseFeature)) in
if selectedFeatures->isEmpty() then null
else
    let selectedFeature : Feature = selectedFeatures->first() in
    let featureValues : FeatureValue = selectedFeature->
        closure(ownedRedefinition.redefinedFeature).ownedMember->
        selectAsKind(FeatureValue) in
    if featureValues->isEmpty() then null
    else featureValues->first().value.evaluate(self)
    endif
```

isSemantic() : Boolean

Check if this `MetadataFeature` has a `metaclass` which is a kind of *`SemanticMetadata`*.

body: `specializesFromLibrary('Metaobjects::SemanticMetadata')`

isSyntactic() : Boolean

Check if this `MetadataFeature` has a `metaclass` that is a kind of *`KerML::Element` (that is, it is from the reflective abstract syntax model).*

body: `specializesFromLibrary('KerML::Element')`

syntaxElement() : Element [0..1]

If this `MetadataFeature` reflectively represents a model element, then return the corresponding `Element instance from the MOF abstract syntax representation of the model.`

pre: `isSyntactic()`

body: `No OCL`

Constraints

checkMetadataFeatureSemanticSpecialization

If this `MetadataFeature` is an application of *`SemanticMetadata`*, then its `annotatingElement` must be a `Type`. The annotated `Type` must then directly or indirectly specialize the specified value of the *`baseType`*, *unless* the `Type` is a `Classifier` and the *`baseType`* represents a kind of `Feature, in which case the Classifier must directly or indirectly specialize each of the types of the Feature.`

```
isSemantic() implies
    let annotatedTypes : Sequence(Type) =
        annotatedElement->selectAsKind(Type) in
    let baseTypes : Sequence(MetadataFeature) =
        evaluateFeature(resolveGlobal(
            'Metaobjects::SemanticMetadata::baseType').
            oclAsType(Feature))->
        selectAsKind(MetadataFeature) in
    annotatedTypes->notEmpty() and
    baseTypes()->notEmpty() and
    baseTypes()->first().isSyntactic() implies
        let annotatedType : Type = annotatedTypes->first() in
        let baseType : Element = baseTypes->first().syntaxElement() in
        if annotatedType.oclIsKindOf(Classifier) and
            baseType.oclIsKindOf(Feature) then
            baseType.oclAsType(Feature).type->
                forAll(t | annotatedType.specializes(t))
        else if baseType.oclIsKindOf(Type) then
            annotatedType.specializes(baseType.oclAsType(Type))
        else
            true
        endif
```

checkMetadataFeatureSpecialization

A `MetadataFeature` must directly or indirectly specialize the base `MetadataFeature` *`Metaobjects::metaobjects`* from the Kernel Semantic Library.

```
specializesFromLibrary("Metaobjects::metaobjects")
```

8.3.4.13 Packages Abstract Syntax

8.3.4.13.1 Overview

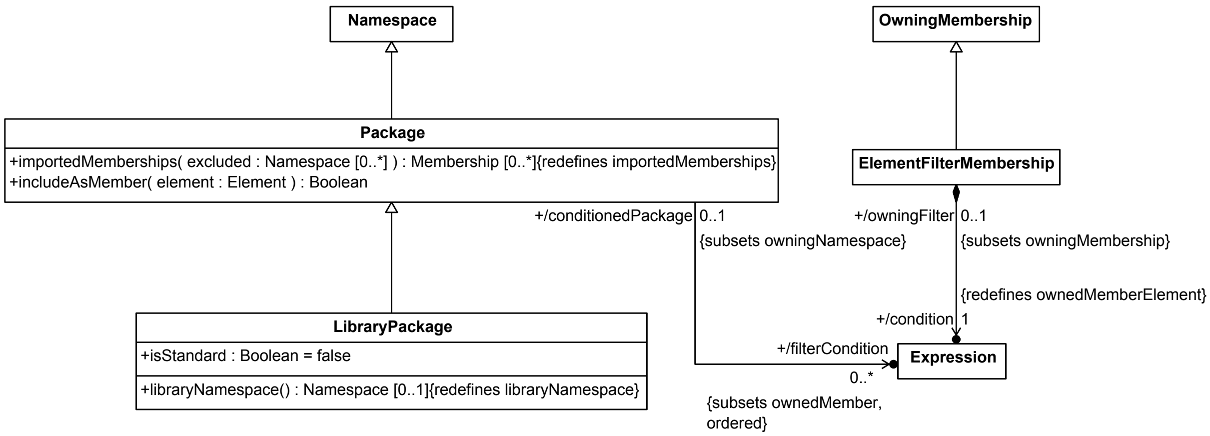

Figure 40. Packages

8.3.4.13.2 ElementFilterMembership

Description

`ElementFilterMembership` is a `Membership` between a `Namespace` and a model-level evaluable *`Boolean`*-valued `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`-valued `Expression` used to filter the imported `members` of the `membershipOwningNamespace` of this `ElementFilterMembership`.

Operations

None.

Constraints

validatePackageElementFilterIsBoolean

The `result` `parameter` of the `condition` `Expression` must directly or indirectly specialize *`ScalarValues::Boolean`*.

```
condition.result.specializesFromLibrary('ScalarValues::Boolean')
```

validatePackageElementFilterIsModelLevelEvaluable

The `condition` `Expression` must be model-level evaluable.

```
condition.isModelLevelEvaluable
```

8.3.4.13.3 LibraryPackage

Description

A `LibraryPackage` is a `Package` that is the container for a model library. A `LibraryPackage` is itself a library `Element` as are all `Elements` that are directly or indirectly contained in it.

General Classes

Package

Attributes

isStandard : Boolean

Whether this `LibraryPackage` contains a standard library model. This should only be set to true for `LibraryPackages` in the standard Kernel Model Libraries or in normative model libraries for a language built on KerML.

Operations

libraryNamespace() : Namespace [0..1]

The `libraryNamespace` for a `LibraryPackage` is itself.

```
body: self
```

Constraints

None.

8.3.4.13.4 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`*-valued `Expression` used to filter the `members` of this `Package`, which are owned by the `Package` are via `ElementFilterMemberships`.

Operations

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 metadataFeatures: Sequence(AnnotatingElement) =
    element.ownedAnnotation.annotatingElement->
        selectByKind(MetadataFeature) in
    self.filterCondition->forAll(cond |
        metadataFeatures->exists(elem |
            cond.checkCondition(elem)))
```

Constraints

derivePackageFilterCondition

The `filterConditions` of a `Package` are the `conditions` of its owned `ElementFilterMemberships`.

```
filterCondition = ownedMembership->
    selectByKind(ElementFilterMembership).condition
```

8.4 Semantics

8.4.1 Semantics Overview

A KerML model is intended to *represent* a system being modeled. The model is *interpreted* to make statements about the modeled system. The model may describe an existing system, in which case, if the model is correct, the statements it is interpreted to make about the system should all be true. A model may also be used to specify an imagined or planned system, in which case the statements the model is interpreted to make should be true for any system that is properly constructed and operated according to the model.

The *semantics* of KerML specify how a KerML model is to be interpreted. The semantics are defined in terms of the abstract syntax representation of the model, and only for models which are *valid* relative to the structure and constraints specified for the KerML abstract syntax (see 8.3). As further specified in this subclause, models expressed in KerML are given semantics by implicitly reusing elements from the semantic models in the Kernel Model Library (see Clause 9). These library models represent conditions on the structure and behavior of the system being modeled, which are further augmented in a user model as appropriate.

A formal specification of semantics allows models to be interpreted consistently. In particular, all KerML models extend library models expressed in KerML itself, understandable by KerML modelers. These library models can then be ultimately reduced to a small, core subset of KerML, which is grounded in mathematical logic. The goal is to provide uniform model interpretation, which improves communication between everyone involved in modeling, including modelers and tool builders.

KerML semantics are specified by a combination of mathematics and model libraries, as illustrated in Fig. 41. The left side of this diagram shows the abstract syntax packages corresponding to the three layers of KerML (see 6.1). The right side shows the corresponding semantic layering.

1. The Root Layer defines the syntactic foundation KerML and, as such, does not have a semantic interpretation relative to the modeled system.
2. The Core Layer is grounded in mathematical semantics, supported by the `Base` package from the Kernel Model Library (see 9.2.2). Subclause 8.4.3 specifies the semantics of the Core layer.
3. The Kernel Layer is given semantics fully through its relationship to the Model Library (see Clause 9). Subclause 8.4.4 specifies the semantics of the Kernel layer .

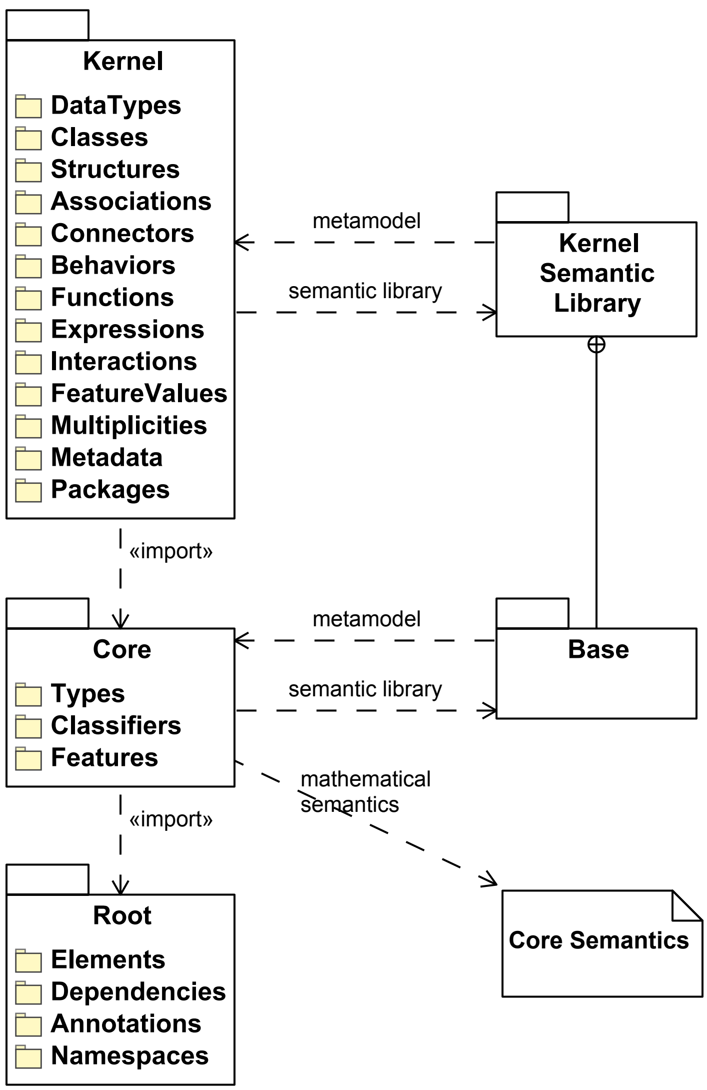

Figure 41. KerML Semantic Layers

8.4.2 Semantic Constraints and Implied Relationships

As described in 8.4.1, KerML semantics are specified by a combination of a mathematical interpretation of the Core layer and a set of required relationships between Core and Kernel model elements and elements of the Kernel Semantic Library (see 9.2). The latter requirements are formalized by *semantic constraints* included in the KerML abstract syntax (see also 8.3.1 on the various kinds of constraints in the abstract syntax). Additionally, other semantic constraints require relationships between elements within a user model necessary for the model to be semantically well formed.

Specifically, there are four categories of semantic constraints, each dealing with a different kind of relationship.

1. *Specialization constraints.* These constraints require that `Type` elements of a certain kind directly or indirectly specialize some specific base `Type` from the Kernel Semantic Library. They are the fundamental means for providing semantics to abstract syntax elements in the Kernel layer. Specialization constraints always have the word `Specialization` in their name. For example, `checkDataTypeSpecialization` requires that a `DataType` directly or indirectly specialize the Semantic Library `DataType` *`Base::DataValue`*.
2. *Redefinition constraints.* These constraints require that certain `Features` in a model have `Redefinition` relationships with certain other `Features` in the model. While `Redefinitions` are kinds of `Specializations`, redefinition constraints differ from the specialization constraints described above in that they are between two elements of a user model, rather than between an element of a user model and an element of a library model. Redefinition constraints always have the word `Redefinition` in their name. For example, `checkConnectorEndRedefinition` requires that the ends of a `Connector` redefine any ends of the `Types` that it specializes.
3. *Type-featuring constraints.* These constraints require that certain `Features` in a model have `TypeFeaturing` relationships with certain other `Types` in the model. They arise at points in a model in which the `OwningMembership` structure is different than the required `Featuring` relationship, so `FeatureMembership` cannot be used. Type-featuring constraints always have the words `TypeFeaturing` in their name. For example, `checkFeatureValueExpressionFeatureTyping` requires that the `value Expression` owned by a `FeatureValue` relationship (a kind of `OwningMembership`) have the same `featuringTypes` as the owning `featureWithValue` of the `FeatureValue`, rather than being featured by the `featureWithValue` itself (as would have been the case for a `FeatureMembership`).
4. *Binding-connector constraints.* These constraints require that `BindingConnectors` exist between certain Features in a model. The primary example of such a constraint is `checkFeatureValueBindingConnector`, which requires that the `featureWithValue` of a `FeatureValue` own a `BindingConnector` between itself and the `result` parameter of the `value Expression` of the `FeatureValue`.

A KerML model parsed from the textual concrete syntax (see 8.2) or obtained through model interchange (see Clause 10) will not necessarily meet the semantic constraints specified for the abstract syntax. In this case, a tool may insert certain *implied* `Relationships` into the model in order to meet the semantic constraints. The overview subclauses for the Core Semantics (see Core Semantics Overview) and Kernel Semantics (see 8.4.4.1) include tables that define what implied `Relationships` should be included to satisfy each semantic constraint when it would otherwise be violated. In all cases, the semantics of a model are only defined if it meets all semantic and validation constraints (see 8.3.1).

When including implied `Relationships` for specialization constraints, it is possible that multiple such constraints may apply to a single element. For example, a `Structure` is a kind of `Class`, which is a kind of `Classifier`, and there are specialization constraints for all three of these metaclasses, with corresponding implied `Subclassification Relationships`. However, simply including all three implied `Subclassification` would be redundant, because the `Subclassification` implied by the `checkStructureSpecialization` constraint will also automatically satisfy the `checkClassSpecialization` and `checkClassifierSpecialization` constraints.

Therefore, in order to avoid redundant `Relationships`, a tool should observe the following rules when selecting which `Specializations` to actually include for a certain `specific Type`, out of the set of those implied by all specialization constraints applicable to the `Type`:

1. If there is any ownedSpecialization or other implied Specialization whose general Type is a direct or indirect subtype of (but not the same as) the general Type of an implied Specialization, or if there is an ownedSpecialization with the same general Type, then that implied Specialization should *not* be included.
2. If there are two implied Specializations with the same general Type, then only one should be included.

Note that the above rules do *not* apply to Redefinitions implied by redefinition constraints, because Redefinition relationships have semantics beyond just basic Specialization.

8.4.3 Core Semantics

8.4.3.1 Core Semantics Overview

8.4.3.1.1 Core Semantic Constraints

The Core semantics are primarily specified mathematically, but the Core metaclasses Type, Classifier, and Feature also have certain semantic constraints (see 8.4.2). Subclause 8.4.3.1.2 describes the general mathematical framework for Core semantics, with specific rules for Types, Classifiers and Features given in 8.4.3.2, 8.4.3.3, and 8.4.3.4, respectively. The following summarizes the corresponding semantic constraints.

The checkTypeSpecialization and checkFeatureSpecialization constraints are actually already implied by the mathematical semantics for Types and Features, but they are included in the abstract syntax so that they can also be reflected syntactically in models by the implied Relationships shown in Table 8. In addition, Table 9 lists the implied Relationships for semantic constraints on the Core metaclass Feature that actually support the semantics of various Kernel-layer constructs, as further described in the Kernel Semantics (8.4.4) subclauses referenced in the table entries for those constraints. In all cases, the source and owningRelatedElement of the Relationship is the Element being constrained, with the target being as given in the last column of the table.

Table 8. Core Semantics Implied Relationships

Semantic Constraint	Implied Relationship	Target
checkTypeSpecialization	Subclassification	*Base::Anything* (see 9.2.2.2.1)
checkFeatureSpecialization	Subsetting	*Base::things* (see 9.2.2.2.7)

Notes

1. The checkTypeSpecialization constraint applies to all Types, but the Subclassification Relationship is only implied for Classifiers (see 8.4.3.3).
2. Satisfaction of the checkFeatureSpecialization constraint implies satisfaction of the checkTypeSpecialization constraint (see 8.4.3.4).

Table 9. Core Semantics Implied Relationships Supporting Kernel Semantics

Semantic Constraint	Implied Relationship	Target
checkFeatureDataValue Specialization	Subsetting	*Base::dataValues* (see 9.2.2.2.3) Supports Data Types Semantics (see 8.4.4.2)
checkFeatureOccurrence Specialization	Subsetting	*Occurrences::occurrences* (see 9.2.4.2.14) Supports Classes Semantics (see 8.4.4.3)

Semantic Constraint	Implied Relationship	Target
checkFeatureSuboccurrence Specialization	Subsetting	*Occurrences::Occurrence::suboccurrences* (see 9.2.4.2.13) Supports Classes Semantics (see 8.4.4.3)
checkFeatureObject Specialization	Subsetting	*Objects::objects* (see 9.2.5.2.8) Supports Structures Semantics (see 8.4.4.4)
checkFeatureSubobject Specialization	Subsetting	*Objects::Object::subobjects* (see 9.2.5.2.7) Supports Structures Semantics (see 8.4.4.4)
checkFeatureEnd Specialization	Subsetting	*Links::Link::participant* (see 9.2.3.2.3) Supports Associations Semantics (see 8.4.4.5)
checkFeatureEndRedefinition	Redefinition	endFeatures of supertypes of the owning Type of the Feature Supports Associations and Connectors Semantics (see 8.4.4.5 and 8.4.4.6)
checkFeatureParameter Redefinition	Redefinition	parameters of supertypes of the owning Behavior or Step of the Feature Supports Behaviors and Steps Semantics (see 8.4.4.7)
checkFeatureResult Redefinition	Redefinition	result parameters of supertypes of the owning Function or Expression of the Feature Supports Functions and Expressions Semantics (see 8.4.4.8)
checkFeatureResult Specialization	FeatureTyping Subsetting	The DataType from *ScalarValues* (see 9.3.2) corresponding to the kind of the owning LiteralExpression or the referent of the owning FeatureReferenceExpression of the Feature Supports Expressions Semantics (see 8.4.4.9)
checkFeatureItemFlowFeature Redefinition	Redefinition	*Transfer::source::sourceOutput* or *Transfer::target::targetInput* (see 9.2.7.2.9) Supports Item Flows Semantics (see 8.4.4.10)

Semantic Constraint	Implied Relationship	Target
`checkFeatureValuation Specialization`	`Subsetting`	The `result` of the `value Expression` of an `owned FeatureValue` of a `Feature` Supports Feature Values Semantics (see 8.4.4.11)

Notes

1. For the `checkFeatureResultSpecialization` constraint, the implied `Specialization` is a `FeatureTyping` if the `owningType` of the `Feature` is a `LiteralExpression` and a `Subsetting` if the `owningType` is a `FeatureReferenceExpression`.

8.4.3.1.2 Core Semantics Mathematical Preliminaries

The mathematical specification of Core semantics uses a model-theoretic approach. Core mathematical semantics are expressed in first order logic notation, extended as follows:

1. A conjunction specifying that multiple variables are members of the same set can be shortened to a comma-delimited series of variables followed by a single membership symbol ($s_1, s_2 \in S$ is short for $s_1 \in S \land s_2 \in S$). Quantifiers can use this in variable declarations, rather than leaving it to the body of the statement before an implication ($\forall t_g, t_s \in V_T$ ... 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 given in 8.1.

The mathematical semantics use the following model-theoretic terms, explained in terms of this specification:

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

The above terms are mathematically defined below.

- 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 8.3.3.1).
  - $V_C \subseteq V_T$ is a set of classifiers (model elements classified by `Classifier` or its specializations, see 8.3.3.2), including at least *`Base::Anything`* from KerML Semantic Model Library, see 9.2.2).
  - $V_F \subseteq V_T$ is a set of features (model elements classified by `Feature` or its specializations, see 8.3.3.3), including at least *`Base::things`* from the KerML Semantic Model Library (see 9.2.2).
  - $V_T = V_C \cup V_F$
- An interpretation $I = (\Delta, \Sigma, \cdot^T)$ for V is a 2-tuple where:
  - Δ is a non-empty set (*universe*),
  - $\Sigma = (P, <_P)$ is a non-empty set P with a strict partial ordering $<_P$ (*marking set*), and

◦ $\cdot^T$ is an (*interpretation*) function relating elements of the vocabulary to sets of all non-empty tuples (*sequences*) of elements of the universe, with an element of the marking set in between each one for sequences of multiple elements. It has domain V_T and co-domain that is the power set of *S,* where

$$S = \{ (d_1) \} \cup \{ (d_1, p_1, d_2) \} \cup \ldots \cup \{ (d_1, p_1, d_2, \ldots p_{i+1}, d_{i+2}) \} \cup \ldots$$
$$\textit{such that } i \in \mathrm{Z}^+, \; d_i \in \Delta, \; p_i \in P$$

The semantics of KerML are restrictions on the interpretation relationship, as given mathematically in this and subsequent subclauses on the Core semantics. The phrase *result of interpreting* a model (vocabulary) element refers to sequences paired with the element by $\cdot^T$, also called the *interpretation* of the model element, for short.

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

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

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 or all of the second starting at the beginning, otherwise is false.

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

- *tail* is true if the first sequence is the same as the second for some or all of the second finishing at the end, otherwise is false:

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

- *head-tail* is true if the first and second sequences are the head and tail of the third sequence, respectively, otherwise is false:

  $$\forall s_1, s_2 \;\; head\text{-}tail(s_1, s_2, s_0) \Rightarrow$$
  $$\texttt{form:Sequence}(s_1) \wedge \texttt{form:Sequence}(s_2) \wedge \texttt{form:Sequence}(s_0)$$
  $$\forall s_1, s_2 \;\; head\text{-}tail(s_1, s_2, s_0) \equiv head(s_1, s_0) \wedge tail(s_2, s_0)$$

- *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.

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

- *concat-around* is true if the first sequence has the second as head, the fourth as tail, and the third element in between.

$$\forall s_0, s_1, p, s_2\ \ concat\text{-}around(s_0, s_1, p, s_2) \Rightarrow \texttt{form:Sequence}(s_0) \land \texttt{form:Sequence}(s_1) \land \texttt{form:Sequence}(s_2)$$

$$\forall s_0, s_1, p, s_2\ \ concat\text{-}around(s_0, s_1, p, s_2) \equiv (length(s_0) = length(s_1) + length(s_2) + 1) \land head\text{-}tail(s_1, s_2, s_0) \land at(p, length(s_1){+}1)$$

- *reverse* is true if the sequences have the same elements, but in reverse order, otherwise is false.

$$\forall s_1, s_2\ \ reverse(s_1, s_2) \Rightarrow \texttt{form:Sequence}(s1) \land \texttt{form:Sequence}(s2)$$

$$\forall s_1, s_2\ \ reverse(s_1, s_2) \equiv (length(s_1) = length(s_2)) \land (\forall i \in Z^+\ \ i \geq 1 \land i \leq length(s_1) \Rightarrow at(s_1, (length(s_1) - i{+}1) = at(s_2, i))$$

8.4.3.2 Types Semantics

Abstract syntax reference: 8.3.3.1

The `checkTypeSpecialization` constraint requires that all `Types` directly or indirectly specialize *`Base::Anything`* (see 9.2.2.2.1). However, there is *no* implied relationship shall be inserted to satisfy this constraint for a `Type` that is not a `Classifier` or a `Feature` (see also 8.4.3.3 and 8.4.3.4 on Classifiers and Features, respectively).

The mathematical interpretation (see8.4.3.1.2) 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`.

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

8.4.3.3 Classifiers Semantics

Abstract syntax reference: 8.3.3.2

The `checkTypeSpecialization` constraint is semantically required for `Classifiers` by the rules below. If necessary, it may be syntactically satisfied in a model by inserting an implied `Subclassification Relationship` to *`Base::Anything`* (see also Table 8).

The mathematical interpretation (see 8.4.3.1.2) 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.

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

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

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

8.4.3.4 Features Semantics

Abstract syntax reference: 8.3.3.3

The `checkFeatureSpecialization` constraint is semantically required by the first two rules below, combined with the definition of $.^T$ in 8.4.3.1.2. If necessary, it may be syntactically satisfied in a model by inserting an implied `Subsetting Relationship` to *`Base::things`* (see also Table 8). Note that satisfaction of the `checkFeatureSpecialization` constraint implies satisfaction of the `checkTypeSpecialization` constraint, because *`Base::things`* is a `FeatureTyping` specialization of *`Base::Anything`*.

The mathematical interpretation (see 8.4.3.1.2) of the `Features` in a model shall satisfy the following rules:

1. The interpretations of `Features` must have length greater than two.

$$\forall s \in S, f \in V_F \;\; s \in (f)^T \Rightarrow length(s) > 2$$

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

$$(\texttt{things})^T = \{\, s \mid s \in S \wedge length(s) > 2 \,\}$$

See other rules below.

`Features` interpreted as sequences of length three or more can be treated as if they were interpreted as ordered triples ("marked" binary relations), where the first and third elements are interpretations of the domain and co-domain of the `Feature`, respectively, while the second element is a *marking* from *P*. 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, with an elements of *P* (*marking*) marking in between them.
- 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`.

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

Markings for the same s_1 above can be related by $<_P$ to order s_2 across multiple interpretations (values) of *f*. Interpretations of *f* can have the same s_1 and s_2, differing only in *p* to distinguish duplicate s_2 (values of *f*).

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

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

$$\begin{aligned}
&\forall s_0 \in S, f \in V_F \;\; s_0 \in (f)^T \;\Rightarrow\; \exists s_t, s_1, s_2 \in S, p \in P \;\; tail(s_t, s_0) \wedge head\text{-}tail(s_1, s_2, s_t) \;\wedge \\
&\quad (length(s_t) > length(s_1) + length(s_2)) \wedge feature\text{-}pair(s_1, p, s_2, f)
\end{aligned}$$

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

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

5. The `multiplicity` of a `Feature` includes the cardinality of its values, counting duplicates.

$$\forall s_1 \in S, f \in V_F, n \in Z^+ \;\; (\forall t_1 \in V_T \;\; t_1 \in f.\texttt{featuringType} \;\Rightarrow\; s_1 \in (t_1)^{minT}) \;\;\wedge$$
$$n = \#\{(p, s_2) \mid \textit{feature-pair}(s_1, p, s_2, f)\} \;\;\Rightarrow$$
$$\exists p \in P \; \textit{feature-pair}(s_1, p, (n), f.\texttt{multiplicity})$$

6. If a `Feature` is unique, there are no values with the same markings.

$$\forall s_1, s_2 \in S, p_1, p_2 \in P, f \in V_F \;\; f.\texttt{isUnique} \;\Rightarrow$$
$$(\textit{feature-pair}(s_1, p_1, s_2, f) \wedge \textit{feature-pair}(s_1, p_2, s_2, f) \;\Rightarrow\; p_1{=}p_2)$$

7. If a `Feature` is ordered, the markings of its values are totally ordered and mark exactly one value each.

$$\forall s_1, s_2, s_3 \in S, p_1, p_2 \in P, f \in V_F \;\; f.\texttt{isOrdered} \;\Rightarrow$$
$$(\textit{feature-pair}(s_1, p_1, s_2, f) \wedge \textit{feature-pair}(s_1, p_2, s_3, f) \;\Rightarrow\; (p_1{=}p_2 \wedge s_2{=}s_3) \vee p_1 <_P p_2 \vee p_2 <_P p_1)$$

8. Sequences in the interpretation of an inverting feature are the reverse of those in the inverted feature.

$$\forall f_1, f_2 \in V_F \;\; f_2 \in f_1.\texttt{invertingFeatureInverting.invertingFeature} \Rightarrow$$
$$(\forall s_1 \in S \;\; s_1 \in (f_1)^T \equiv \; (\exists s_2 \in S \;\; s_2 \in (f_2)^T \wedge \textit{reverse}(s_2, s_1)))$$

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

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

The interpretations of a `Feature` (f) specified as a chain of two others (f_1 and f_2) are all sequences formed from `Feature` pairs of the two others that share the same sequence as second and first in their pairs, respectively. If f is ordered, marking order in interpretations of f applies the order of f_1 values to those of f_2 found via each value of f_1. If f is non-unique, duplicate values of f_2 (which might be due to multiple values of f_1) are preserved in f, otherwise f_2 can have no duplicate values (including any due to multiple values of f_1).

$$\forall paths, sd, f_1, f_2, scd \;\; paths = all\text{-}chain\text{-}path\text{-}2(sd, f_1, f_2, scd) \Rightarrow$$
$$\texttt{form:Set}(paths) \wedge sd, scd \in S \wedge f_1, f_2 \in V_F$$
$$\forall sd, f_1, f_2, scd \;\; all\text{-}chain\text{-}path\text{-}2(sd, f_1, f_2, scd) =$$
$$\{ (pm, sm, pm11) \mid pm, pm11 \in P \wedge sm \in S \wedge$$
$$feature\text{-}pair(sd, pm, sm, f_1) \wedge feature\text{-}pair(sm, pm11, scd, f_2) \}$$

$$\forall f, f_1, f_2 \;\; chain\text{-}feature\text{-}2(f, f_1, f_2) \Rightarrow f, f_1, f_2 \in V_F$$

$$\forall f, f_1, f_2 \;\; chain\text{-}feature\text{-}2(f, f_1, f_2) \Rightarrow$$
$$(\forall sd, scd \in S \;\; \#\{ pcd \mid feature\text{-}pair(sd, pcd, scd, f) =$$
$$\#all\text{-}chain\text{-}path\text{-}2(f_1, f_2, scd))$$

$$\forall f, f_1, f_2 \;\; chain\text{-}feature\text{-}2(f, f_1, f_2) \Rightarrow$$
$$(\forall sd, scd_1, scd_2, ppath_1, ppath_2 \wedge$$
$$ppath_1 \in all\text{-}chain\text{-}path\text{-}2(f_1, f_2, scd_1) \wedge$$
$$ppath_2 \in all\text{-}chain\text{-}path\text{-}2(f_1, f_2, scd_2) \wedge$$
$$(\forall pm_1, pm_{11} \in P, sm_1, sm_2 \in S$$
$$pm_1{=}at(ppath_1, 1) \wedge sm_1{=}at(ppath_1, 2) \wedge pm_{11}{=}at(ppath_1, 3) \wedge$$
$$pm_2{=}at(ppath_2, 1) \wedge sm_2{=}at(ppath_2, 2) \wedge pm_{21}{=}at(ppath_2, 3) \wedge$$
$$((pm_1 <_P pm_2) \vee (pm_1{=}pm_2 \wedge sm_1{=}sm_2 \wedge pm_{11} <_P pm_{21}) \Rightarrow$$
$$(\exists pcd_1, pcd_2 \in P \;\; pcd_1 <_P pcd_2 \wedge$$
$$feature\text{-}pair(sd, pcd_1, scd_1, f) \wedge feature\text{-}pair(sd, pcd_2, scd_2, f))))$$

A Feature (*f*) specified as a chain of two or more others (*fl*, a list of features longer than 1) is the last in a series of `Features` specified by incremental subchains (*flc*), starting with the first two `Features` in *fl*, (specifying the first `Feature` in *flc*), then the first three in *fl* (specifying the second `Feature` in *flc*), and so on, to all the `Features` in *fl* (specifying the last feature in *flc*, which is the original `Feature` *f*). If *f* is ordered, marking order in interpretations of each subchain apply to values of later subchains. If *f* is non-unique, duplicate values of the last `Feature` in *fl* (which might be due to multiple values of the other `Features`) are preserved in *f*, otherwise the last `Feature` in *fl* can have no duplicates (including any due to multiple values of the other `Features`).

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

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

8.4.4 Kernel Semantics

8.4.4.1 Kernel Semantics Overview

The semantics of constructs in the Kernel Layer are specified in terms of the foundational constructs defined in the Core layer supported by reuse of model elements from the Kernel Semantic Model Library (see 9.2). The most common way in which library model elements are used is through specialization, in order to meet subtyping constraints specified in the abstract syntax. For example, `Classes` are required to (directly or indirectly) subclassify *`Object`* from the *`Objects`* library model, while `Features` typed by `Classes` must subset *`objects`*. Similarly, `Behaviors` must subclassify *`Performance`* from the *`Performances`* library model, while `Steps` (`Features` typed by `Behaviors`) must subset *`performances`*. The requirement for such specialization is specified by specialization constraints in the abstract syntax, as listed in Table 10 along with the implied Specializations that may be used to satisfy them (see 8.4.2 for discussion of specialization constraints and implied Relationships).

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*. Such patterns are specified by redefinition constraints and other kinds of semantic constraints in the abstract syntax, as listed in Table 11 along with the implied Relationships that may be used to satisfy them (see also 8.4.2). In addition the Core semantic constraints listed in Table 9 actually support the semantics of Kernel layer constructs.

In all cases, all Kernel syntactic constructs can be ultimately reduced to semantically equivalent Core patterns. Various elements of the Kernel abstract syntax essentially act as "markers" for modeling patterns typing the Kernel to the Core. The following subclauses specify the semantics for each syntactic area of the Kernel Layer in terms of the semantic constraints that must be satisfied for various Kernel elements, the pattern of relationships these imply, and the model library elements that are reused to support this.

Table 10. Kernel Semantics Implied Specializations

Semantic Constraint	Implied Relationship	Target
checkDataTypeSpecialization	Subclassification	*Base::DataValue* (see 9.2.2.2.2)
checkClassSpecialization	Subclassification	*Occurrences::Occurrence* (see 9.2.4.2.13)
checkStructureSpecialization	Subclassification	*Objects::Object* (see 9.2.5.2.7)
checkAssociation Specialization	Subclassification	*Links::Link* (see 9.2.3.2.3)
checkAssociationBinary Specialization	Subclassification	*Links::BinaryLink* (see 9.2.3.2.1)
checkAssociationStructure Specialization	Subclassification	*Objects::LinkObject* (see 9.2.5.2.5)
checkAssociationStructure BinarySpecialization	Subclassification	*Objects::BinaryLinkObject* (see 9.2.5.2.1)
checkConnectorSpecialization	Subsetting	*Links::links* (see 9.2.3.2.4)
checkConnectorBinary Specialization	Subsetting	*Links::binaryLinks* (see 9.2.3.2.2)
checkConnectorObject Specialization	Subsetting	*Objects::linkObjects* (see 9.2.5.2.6)
checkConnectorBinaryObject Specialization	Subsetting	*Objects::binaryLinkObjects* (see 9.2.5.2.2)
checkBindingConnector Specialization	Subsetting	*Links::selfLinks* (see 9.2.3.2.6)
checkSuccession Specialization	Subsetting	*Occurrences::happensBeforeLinks* (see 9.2.4.2.2)
checkBehaviorSpecialization	Subclassification	*Performances::Performance* (see 9.2.6.2.13)

Semantic Constraint	Implied Relationship	Target
checkStepSpecialization	Subsetting	*Performances::performances* (see 9.2.6.2.14)
checkStepEnclosedPerformanceSpecialization	Subsetting	*Performances::Performance::enclosedPerformance* (see 9.2.6.2.13)
checkStepSubperformanceSpecialization	Subsetting	*Performances::Performance::subperformance* (see 9.2.6.2.13)
checkStepOwnedPerformanceSpecialization	Subsetting	*Objects::Object::ownedPerformance* (see 9.2.5.2.7)
checkFunctionSpecialization	Subclassification	*Performances::Evaluation* (see 9.2.6.2.3)
checkPredicateSpecialization	Subclassification	*Performances::BooleanEvaluation* (see 9.2.6.2.1)
checkExpressionSpecialization	Subsetting	*Performances::evaluations* (see 9.2.6.2.4)
checkBooleanExpressionSpecialization	Subsetting	*Performances::booleanEvaluations* (see 9.2.6.2.2)
checkInvariantSpecialization	Subsetting	*Performances::trueEvaluations* (see 9.2.6.2.16), for true Invariants, or *Performances::falseEvaluations* (see 9.2.6.2.5), for false (negated) Invariants
checkNullExpressionSpecialization	Subsetting	*Performances::nullEvaluations* (see 9.2.6)
checkLiteralExpressionSpecialization	Subsetting	*Performances::literalEvaluations* (see 9.2.6)
checkOperatorExpressionSpecialization	FeatureTyping	The library Function named by the operator of the OperatorExpression
checkMetadataAccessExpressionSpecialization	Subsetting	*Performances::metadataAccessEvaluations* (see 9.2.6.2.10)
checkItemFlowSpecialization	Subsetting	*Transfers::transfers* (see 9.2.7.2.11), or *Transfers::flowTransfers* (see 9.2.7.2.4), if the ItemFlow has ItemFlowEnds
checkSuccessionItemFlowSpecialization	Subsetting	*Transfers::flowTransfersBefore* (see 9.2.7.2.5)
checkMultiplicitySpecialization	Subsetting	*Base::naturals* (see 9.2.2.2.5)

Semantic Constraint	Implied Relationship	Target
`checkMetaclassSpecialization`	`Subclassification`	*`Metaobjects::Metaobject`* (see 9.2.16.2.1)
`checkMetadataFeature Specialization`	`Subsetting`	*`Metaobjects::metaobjects`* (see 9.2.16.2.2)
`checkMetadataFeatureSemantic Specialization`	`Specialization` `Subclassification` `FeatureTyping` `Subsetting` (see Note 2)	(See Note 2 and 8.4.4.13)

Notes.

1. For all constraints *other* than `checkMetadataFeatureSemanticSpecialization`, the `source` of any implied `Relationship` is the annotated element of the constraint, with the `target` as given in the table.
2. The `checkMetadataFeatureSemanticSpecialization` constraint only applies to a `MetadataFeature` that has a `metaclass` that is a kind of *`SemanticMetadata`* (see 9.2.16.2.3). The source of the implied `Relationship` for this constraint is *not* the `MetadataFeature` but, rather, the `Type` annotated by the `MetadataFeature`, and a conforming tool need only insert the `Relationship` if the `MetadataFeature` is an `ownedMember` of the `Type`. The kind of `Relationship` that is implied and its `target` are determined as follows:
    - If the annotated `Type` and the *`baseType`* are both `Classifiers`, then `Subclassification` targeting the *`baseType`*.
    - If the annotated `Type` is a `Feature` and the *`baseType`* is a `Classifier`, then `FeatureTyping` targeting the *`baseType`*.
    - If the annotated `Type` and the *`baseType`* are both `Features`, then `Subsetting` targeting the *`baseType`*.
    - If the annotated `Type` is a `Classifier` and the *`baseType`* is a `Feature`, then `Subclassifications` targeting each of the types of the `Feature`.

Table 11. Kernel Semantics Other Implied Relationships

Semantic Constraint	Implied Relationship	Target (or source and target for binding)
`checkItemFeature Redefinition`	`Redefinition`	*`Transfers::Transfer::item`* (see 9.2.7.2.9)
`checkFeatureItemFlowFeature Redefinition`	`Redefinition`	*`Transfer::source:: sourceOutput`* or *`Transfer::target:: targetInput`* (see 9.2.7.2.9)
`checkConnectorTypeFeaturing` (see Note 2)	`TypeFeaturing`	The common directly or indirectly featuring `Type` for the `Connector` and its `relatedElements`
`checkExpressionTypeFeaturing`	`TypeFeaturing`	The `featuringTypes` of the `featureWithValue` of the `FeatureValue` that owns the `Expression`

Semantic Constraint	Implied Relationship	Target (or source and target for binding)
`checkFunctionResult BindingConnector`	`BindingConnector`	From the `result` of the result `Expression` of the `Function` to its `result` `parameter`.
`checkExpressionResult BindingConnector`	`BindingConnector`	From the `result` of the result `Expression` of the constrained `Expression` to its `result` `parameter`.
`checkFeatureReference ExpressionBindingConnector`	`BindingConnector`	Between the `referent` and `result` of the `FeatureReferenceExpression`
`checkInvocationExpression ConstructorBindingConnector` (see Note 3)	`BindingConnector`	Between the `InvocationExpression` itself and its `result` `parameter`.
`checkInvocationExpression DefaultValueBindingConnector` (see Note 4)	`BindingConnector`	Between `features` of the `InvocationExpression` and `results` of default `value` `Expressions` for those `features`.
`checkFeatureChainExpression TargetRedefinition` (see Note 5)	`Redefinition`	*`ControlFunctions::'.'::source::target`* (see 9.4.17)
`checkFeatureChainExpression SourceTargetRedefinition` (see Note 5)	`Redefinition`	The `targetFeature` of the `FeatureChainExpression`
`checkFeatureValue BindingConnector`	`BindingConnector`	Between the `featureWithValue` of the `FeatureValue` and the `result` of its `value` `Expression`

Notes

1. For redefinition and type featuring constraints, the annotated element of the constraint is the `source` and `owningRelatedElement` of the implied `Relationship`, with the `target` as given in the last column table. For binding connector constraints, the annotated element of the constraint is the `owningNamespace` of the implied `Relationship`, with the `source` and `target` of the `Relationship` as given in the last column of the table.
2. For the `checkConnectorTypeFeaturing` constraint, an implied `TypeFeaturing` shall only be included to satisfy if the `Connector` has no `owningType` and no `ownedTypeFeaturings`.
3. The `checkInvocationExpressionConstructorBindingConnector` constraint only applies if the invoked type is not a `Behavior` or a `Feature` typed by a `Behavior`.
4. The `checkInvocationExpressionDefaultValueBindingConnector` constraint applies to each `feature` of an `InvocationExpression` that redefines a `Feature` for which there is an *effective default value* (see 8.4.4.11).
5. For the `checkFeatureChainExpressionTargetRedefinition` and `checkFeatureChainExpressionSourceTargetRedefinition` constraints, the `redefiningFeature` of the implied `Redefinition` is a nested `Feature` of the first owned input

parameter of the FeatureChainExpression (corresponding to the *source* parameter of the '.' Function).

8.4.4.2 Data Types Semantics

Abstract syntax reference: 8.3.4.1

The checkDataTypeSpecialization constraint requires that DataTypes specialize the base DataType *Base::DataValue* (see 9.2.2.2.2). The checkFeatureDataValueSpecialization constraint requires that Features typed by a DataType specialize the Feature *Base::dataValues* (see 9.2.2.2.3), which is typed by *Base::DataValue*.

```
datatype D specializes Base::DataValue {
    feature a : ScalarValue::String subsets Base::dataValues;
    feature b : D subsets Base::dataValues;
}
```

The Type *Base::DataValue* is disjoint with *Occurrences::Occurrence* and *Links::Link*, the base Types for Classes and Associations (see 8.4.4.3 and 8.4.4.5, respectively). This means that a DataType cannot specialize a Class or Association and that a Feature typed by a DataType cannot also be typed by a Class or Association.

8.4.4.3 Classes Semantics

Abstract syntax reference: 8.3.4.2

The checkClassSpecialization constraint requires that Classes specialize the base Class *Occurrences::Occurrence* (see 9.2.4.2.13). The checkFeatureOccurrenceSpecialization constraint requires that Features typed by a Class specialize the Feature *Occurrences::occurrences* (see 9.2.4.2.14), which is typed by *Occurrences::Occurrence*. Further, the checkFeatureSuboccurrenceSpecialization constraint requires that composite Features typed by a Class, and whose ownedType is a Class or another Feature typed by a Class, specialize the Feature *Occurrences::Occurrence::suboccurrences* (see 9.2.4.2.13), which subsets *Occurrences::occurrences*.

```
class C specializes Occurrences::Occurrence {
    feature a : C subsets Occurrences::occurrences;
    composite feature b : C subsets Occurrences::Occurrence::suboccurrences;
}
```

The Class *Occurrences::Occurrence* is disjoint with *Base::DataValues*, the base Type for DataTypes (see 8.4.4.2). This means that a Class cannot specialize a DataType and that a Feature typed by a Class cannot also be typed by a DataType. Note that *Occurrences::Occurrence* is *not* disjoint with *Link::Links*, because an AssociationStructure is both an Association and a Structure (which is a kind of Class), so the base AssociationStructure *Objects::LinkObject* specializes both *Link::Links* and (indirectly) *Occurrences::Occurrence*.

Unlike *DataValues*, *Occurrences* are modeled as occurring in three-dimensional space and persisting over time. The *Occurrences* library model includes an extensive set of Associations between *Occurrences* that model various spatial and temporal relations, such as *InsideOf, OutsideOf, HappensBefore, HappensDuring,* etc. In particular, the Association *HappensBefore* is the base Type for Successions, the basic modeling construct for time-ordering *Occurrences* (see 8.4.4.6 on the semantics of Successions). For further detail on the *Occurrences* model, see 9.2.4.1.

8.4.4.4 Structures Semantics

Abstract syntax reference: 8.3.4.3

The `checkStructureSpecialization` constraint requires that `Structures` specialize the base `Structure` *`Objects::Object`* (see 9.2.5.2.7). The `checkFeatureObjectSpecialization` constraint requires that `Features` typed by a `Structure` specialize the `Feature` *`Objects::objects`* (see 9.2.5.2.8), which is typed by *`Objects::Object`*. Further, the `checkFeatureSubobjectSpecialization` constraint requires that composite `Features` typed by a `Structure`, and whose `ownedType` is a `Structure` or another `Feature` typed by a `Structure`, specialize the `Feature` *`Objects::Object::subobjects`* (see 9.2.5.2.7), which subsets *`Object::objects`*.

```
struct S specializes Objects::Object {
    feature a : S subsets Object::objects;
    composite feature b : S subsets Objects::Object::subobjects;
}
```

`Objects` are *`Occurrences`* representing physical or virtual structures that occur over time. For physical structures, the *`Objects`* library model also provides a the specialization *`StructuredSpaceObject`*, which models *`Objects`* that can be spatial decomposed into cells of the same or lower dimension. The `Type` *`Object`* is disjoint with the `Type` *`Performance`*, another specialization of *`Occurrence`*, which is the base `Type` for `Behaviors` (see 8.4.4.7 on the semantics of `Behaviors`). For further detail on the *`Objects`* model, see 9.2.5.1.

8.4.4.5 Associations Semantics

Abstract Syntax Reference: 8.3.4.4

8.4.4.5.1 Associations

The `checkAssociationSpecialization` and `checkFeatureEndSpecialization` constraints require that an `Association` specialize the base `Association` *`Links::Link`* (see 9.2.3.2.3) and that its `associationEnds` subset *`Links::Link::participant`*. These constraints essentially require an N-ary `Association` to have the form (with implied relationships included):

```
assoc 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 *`Link`* instance for an `Association` is thus a tuple of *`participants`*, where each `participant` is a *single* value of an `associationEnd` of the `Association`.

As `endFeatures`, the `associationEnds` of an `Association` are given a special semantics compared to other `Features`. Even if an `associationEnd` has a declared `multiplicity` other than `1..1`, the `associationEnd` is required to effectively have multiplicity `1..1` as a `participant` in the *`Link`*. 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 declared `multiplicity` 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*.

The checkFeatureEndRedefinition constraint requires that, if an Association has an ownedSubclassification to another Association, then its associationEnds redefine the associationEnds of the superclassifier Association. In this case, the subclassifier Association will indirectly specialize *Link* through a chain of Subclassifications, and each of its associationEnds will indirectly subset *Links::participant* through a chain of redefinitions and a subsettings.

The checkAssociationBinarySpecialization constraint requires that a binary Association (one with exactly two associationEnds) specialize *Links::BinaryLink*. *BinaryLink* specializes *Link* to have exactly two *participants* corresponding to two ends called *source* and *target*. As required by the checkFeatureEndRedefinition constraint, the first associationEnd of a binary Association will redefine *Links::BinaryLink::source* and its second associationEnd will redefine *Links::BinaryLink::target*.

```
assoc B specializes Links::BinaryLink {
    end feature e1 redefines Links::BinaryLink::source;
    end feature e2 redefines Links::BinaryLink::target;
}
```

8.4.4.5.2 Association Structures

An AssociationStructure is both an Association and a Structure and, therefore, the semantic constraints of both Associations and Structures (see 8.4.4.4) apply to AssociationStructures. The checkAssociationStructureSpecialization constraint requires an AssociationStructure to specialize *Objects::LinkObject* (see 9.2.5.2.5), which specializes both *Links::Link* and *Objects::Object*. The checkAssociationStructureBinarySpecialization constraint requires a binary AssociationStructure to specialize *Objects::BinaryLinkObject* (see 9.2.5.2.1), which specializes both *Links::BinaryLink* and an *Objects::LinkObject*.

8.4.4.6 Connectors Semantics

Abstract syntax reference: 8.3.4.5

8.4.4.6.1 Connectors

A Connector can only be typed by Associations. The checkConnectorSpecialization constraint then requires that Connectors specialize the base Feature *Link::links* (see 9.2.3.2.4), which is typed by the base Association *Links::Link* (see 9.2.3.2.3). Further, the checkFeatureEndRedefinition constraint requires that the connectorEnds of a Connector redefine the associationEnds of its typing Associations. As a result, a Connector typed by an N-ary Association is essentially required to have the form (with implicit relationships included):

```
connector a : A subsets Links::links {
    end feature e1 redefines A::e1 references f1;
    end feature e2 redefines A::e2 references f2;
    ...
    end feature eN redefines A::eN references fN;
}
```

where *e1*, *e2*, ..., *eN* are the names of associationEnds of the Association *A*, in the order they are defined in *A*, and the *f1*, *f2*, ..., *fN* are the relatedFeatures of the Connector. Multiplicities declared for connectorEnds have the same special semantics as for associationEnds (see 8.4.4.5). If *A* is an AssociationStructure, then the checkConnectorObjectSpecialization constraint requires that the Connector subsets *Objects::linkObjects* (see 9.2.5.2.6) instead of *Links::link*.

A binary Connector is a Connector with exactly two connectorEnds, that is, a Connector typed by a binary Association. The checkConnectorBinarySpecialization constraint requires that binary Connectors specialize the base Feature *Link::binaryLinks* (see 9.2.3.2.2), which is typed by the Association *Links::BinaryLink* (see 9.2.3.2.1). 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*.

```
connector b : B subsets Links::binaryLinks {
    end feature source redefines B::source references f1;
    end feature target redefines B::target references f2;
}
```

If *B* is an AssociationStructure, then the checkConnectorBinaryObjectSpecialization constraint requires that the Connector subsets *Objects::binaryLinkObjects* (see 9.2.5.2.2) instead of *Links::binaryLinks*.

A Connector therefore specifies a subset of the *Links* of its typing Associations for which the *participants* are values of the relatedFeatures of the Connector. In addition, the checkConnectorTypeFeaturing constraint requires that the featuringTypes of a Connector be consistent with those of its relatedFeatures. Typically, a Connector will have an owningType that is its featuringType, in which case all of its relatedFeatures must also be featured in the context of this Type. An implicit TypeFeaturing may be included to satisfy the checkConnectorTypeFeaturing constraint, but *only* if the Connector has no explicit owningType or ownedTypeFeaturings. The primary case in which an implicit TypeFeaturing is necessary is for a BindingConnector that is itself added implicitly for a FeatureValue (see 8.4.4.11).

```
// This is the simplest case of a Connector satisfying checkConnectorTypeFeaturing,
// in which the Connector and its relatedFeatures all have the same owningType.
classifier C {
    feature f1;
    feature f2;
    connector b subsets Links::binaryLinks {
        end feature references f1 redefines Links::BinaryLink::source;
        end feature references f2 redefines Links::BinaryLink::target;
    }
}
```

8.4.4.6.2 Binding Connectors

The `checkBindingConnectorSpecialization` constraint requires that `BindingConnectors` specialize the `Feature` *`Links::selfLinks`* (see 9.2.3.2.6), which is typed by the `Association` *`SelfLink`* (see 9.2.3.2.5). *`SelfLink`* has two `associationEnds` that subset each other, meaning they identify the same things (have the same values), which then also applies to `BindingConnector` `connectorEnds` that redefine the `associationEnds` of *`SelfLink`*. Since both `associationEnds` of *`SelfLink`* have multiplicity `1..1`, both `connectorEnds` of a `BindingConnector` do also. The general semantic constraints for `Connectors` also apply to `BindingConnectors`.

Thus, a `BindingConnector` declaration of the form

```
binding f1 = f2;
```

is, with implied `Relationships` included, semantically equivalent to

```
connector subsets Links::selfLinks {
    end feature thisThing redefines Links::SelfLink::thisThing references f1;
    end feature thatThing redefines Links::SelfLink::thatThing references f2;
}
```

8.4.4.6.3 Successions

The `checkSuccessionSpecialization` constraint requires that `Successions` specialize the `Feature` *`Occurrences::happensBeforeLinks`* (see 9.2.4.2.2), which is typed by the Association *`HappensBefore`* (see 9.2.4.2.1). *`HappensBefore`* (see 9.2.4.2.1) has two `associationEnds`, asserting that the *`Occurrence`* identified by its first `associationEnd` (`earlierOccurrence`) temporally precedes the one identified by its second (`laterOccurrence`), which then also applies to `Succession` `connectorEnds` that redefine the `associationEnds` of *`HappensBefore`*. The general semantic constraints for `Connectors` also apply to `Successions`.

This, a `Succession` declaration of the form

```
succession first f1 then f2;
```

is, with implied `Relationships` included, semantically equivalent to

```
connector subsets Occurrences::happensBeforeLinks {
    end feature earlierOccurrence references f1
        redefines Occurrences::HappensBefore::earlierOccurrence;
    end feature laterOccurrence references f2
        redefines Occurrences::HappensBefore::laterOccurrence;
}
```

8.4.4.7 Behaviors Semantics

Abstract syntax reference: 8.3.4.6

8.4.4.7.1 Behaviors

The `checkBehaviorSpecialization` constraint requires that `Behaviors` specialize *`Performances::Performance`* (see 9.2.6.2.13). In addition, the `checkFeatureParameterRedefinition` constraint requires that any owned `parameters` (i.e., directed `ownedFeatures`) of a `Behavior` redefine corresponding `parameters` of any other `Behaviors` it specializes.

```
behavior B specializes Performances::Performance {
    in feature x[0..*] subsets Base::things;
    out feature y[0..1] subsets Base::things;
    inout feature z subsets Base::things;
}
behavior B1 specializes B {
    in feature x1[1] redefines B::x;
    out feature y1[1] redefines B::y;
    // z is inherited without redefinition
}
```

8.4.4.7.2 Steps

The `checkStepSpecialization` constraint requires that `Steps` specialize *`Performances::performances`* (see 9.2.6.2.14). In addition, the `checkFeatureParameterRedefinition` constraint requires that any owned `parameters` (i.e., directed `ownedFeatures`) of a `Step` redefine corresponding `parameters` of any other `Steps` or `Behaviors` it specializes. In particular, a `Step` explicitly typed by a `Behavior` will generally redefine the `parameters` of that `Behavior`.

```
step b : B subsets Performances::performances {
    in feature x redefines B::x = x1;
    out feature y redefines B::y;
    inout feature z redefines B::z := z1 ;
}

step b1 : B1 subsets b {
    in feature x redefines B1::x, b::x;
    out feature y redefines B2::y, b::y;
}
```

Further, the `checkStepEnclosedPerformanceSpecialization` and `checkStepSubperformanceSpecialization` constraints require that a `Step` whose `owningType` is a `Behavior` or another `Step` specialize *`Performances::Performance::enclosedPerformance`* or, if it is composite, *`Performances::Performance::subperformance`* (see 9.2.6.2.13). Finally, the `checkStepOwnedPerformanceSpecialization` constraint requires that a composite `Step` whose `owningType` is a `Structure` or a `Feature` typed by a `Structure` specialize *`Objects::Object::ownedPerformance`* (see 9.2.5.2.7).

```
step s subsets Performances::performances {
    step s1 subsets Performances::Performance::enclosedPerformance;
    composite step s2 subsets Performances::Performance::subperformance;
}
struct S specializes Objects::Object {
    composite step ss subsets Objects::Object::ownedPerformance;
}
```

8.4.4.8 Functions Semantics

Abstract syntax reference: 8.3.4.7

8.4.4.8.1 Functions and Predicates

`Functions` are kinds of `Behaviors`. The `checkFunctionSpecialization` constraint requires that `Functions` specialize the base `Function` *`Performances::Evaluation`* (see 9.2.6.2.3), which is a specialization of *`Performances::Performance`*. All other semantic constraints on `Behaviors` (see 8.4.4.7) also apply to `Functions`. In addition, the `checkFeatureResultRedefinition` constraint requires that the `result` parameter of a Function always redefine the result of any its supertypes that are also Functions, regardless of their parameter position.

```
function F specializes Performances::Evaluation {
    in a;
    in b;
    return result redefines Performances::Evaluation::result;
}
function G specializes F {
    in a redefines F::a;
    return result redefines F::result;
    in b redefines F::b;
}
```

Further, if a Function owns an Expression via a ResultExpressionMembership, then the checkFunctionResultBindingConnector constraint requires that the Function have, as an ownedFeature, a BindingConnector between the result parameter of the Expression and the result parameter of the Function.

```
function H specializes Performances::Evaluation {
    return redefines Performances::Evaluation::result;
    binding result = resultExpr.result; // Implied
    resultExpr
}
```

where *resultExpr* is an arbitrary Expression and *resultExpr.result* represents a Feature chain to the Expression result.

A Predicate is a kind of Function, so all semantic constraints for Functions also apply to Predicates. In addition, the checkPredicateSpecialization constraint requires that Predicates specialize the base Predicate *Performances::BooleanEvaluation* (see 9.2.6.2.1), which is a specialization of *Performances::Evaluation*. *BooleanEvaluation* has a result parameter typed by Boolean, so Predicates always have a Boolean result.

```
predicate P specializes Performances::BooleanEvaluation {
    in x : ScalarValues::Real;
    return redefines Performances::BooleanEvaluation::result;
    x > 0
}
```

8.4.4.8.2 Expressions and Invariants

Expressions are kinds of Steps. The checkExpressionSpecialization constraint requires that Expressions specialize the base Expression *Performances::evaluation*s (see 9.2.6.2.3), which is a specialization of *Performances::performances*. All other semantic constraints on Steps (see 8.4.4.7) also apply to Functions. In addition, the checkFeatureResultRedefinition constraint requires that the result parameter of an Expression always redefine the result of any its supertypes that are Functions or other Expressions, regardless of their parameter position.

```
expr f : F subsets Performances::evaluations {
    in a redefines F::a;
    in b redefines F::b;
    return result redefines F::result, Performances::evaluations::result;
}
expr g : G subsets f {
```

```
        return result redefines G::result, f::result;
    }
```

Further, if an `Expression` owns another `Expression` via a `ResultExpressionMembership`, then the `checkExpressionResultBindingConnector` constraint requires that the `Expression` have, as an `ownedFeature`, a `BindingConnector` between the `result` `parameter` of the owned `Expression` and the `result` `parameter` of the owning `Expression`.

```
    expr h subsets Performances::Evaluation {
        binding result = resultExpr.result; // Implied
        resultExpr
    }
```

where *`resultExpr`* is an arbitrary `Expression` and *`resultExpr.result`* represents a `Feature` chain to the `Expression` `result`.

A `BooleanExpression` is a kind of `Expression`, so all semantic constraints for `Expressions` also apply to `BooleanExpressions`. In addition, the `checkBooleanExpressionSpecialization` constraint requires that `BooleanExpressions` specialize the base `BooleanExpression` *`Performances::booleanEvaluations`* (see 9.2.6.2.2), which is a specialization of *`Performances::evaluation`*s.

```
    expr p : P subsets Performances::booleanEvaluations {
        in x : ScalarValues::Integer redefines P::x;
        return redefines P::x, Performance::BooleanEvaluation::result;
    }
```

An `Invariant` is a kind of `BooleanExpression`, so all semantic constraints for `BooleanExpressions` also apply to `Invariants`. In addition, the `checkInvariantSpecialization` constraint requires that `Invariants` specialize *either* the `BooleanExpression` *`Performances::trueEvaluations`* (see 9.2.6.2.2) or, if the `Invariant` is negated, the `BooleanExpression` *`Performances::falseEvaluations`* (see 9.2.6.2.2), both of which are specializations of *`Performances::booleanEvaluations`*. The `BooleanExpression` *`trueEvaluations`* has its `result` bound to `true`, while the `BooleanExpression` *`falseEvaluations`* has its `result` bound to `false`.

```
    inv true i1 subsets Performances::trueEvaluations {
        p(3)
    }
    inv false i2 subsets Performances::falseEvaluations {
        p(-3)
    }
```

8.4.4.9 Expressions Semantics

Abstract syntax reference: 8.3.4.8

8.4.4.9.1 Null Expressions

The `checkNullExpressionSpecialization` constraint requires that `NullExpressions` specialize the `Expression` *`Performances::nullEvaluations`* (see 9.2.6.2.12), which is typed by the `Function` *`Performances::NullEvaluation`* (see 9.2.6.2.11). The `result` `parameter` of *`NullEvaluation`* has multiplicity `0..0`, which means that a `NullExpression` always produces an empty result. The general semantic constraints for `Expressions` (see 8.4.4.8) also apply to `NullExpressions`.

8.4.4.9.2 Literal Expressions

The checkLiteralExpressionSpecialization constraint requires that LiteralExpressions specialize the Expression *Performances::literalEvaluations* (see 9.2.6.2.8), which is typed by the Function *Performances::LiteralEvaluation* (see 8.3.4.8.7). The result parameter of *LiteralEvaluation* has multiplicity 1..1 and is typed by *Base::DataValue* (see 9.2.2.2.2). This means that a LiteralExpression always produces a single *DataValue* as its result. What value is actually produced depends on the kind of LiteralExpression. The general semantic constraints for Expressions (see 8.4.4.8) also apply to LiteralExpressions.

With the exception of LiteralInfinity, each kind of LiteralExpression has a value property typed by a UML primitive type [UML, MOF]. The result produced by such a LiteralExpression is given by this value. LiteralInfinity does not have a value property, because its result is always "infinity" (written * the KerML textual notation; see 8.2.5.8.4), which is a number from the DataType *ScalarValues::Positive* that is greater than all the integers. The checkFeatureResultSpecialization requires that the result parameter of a LiteralExpression have the corresponding DataType from the KerML *ScalarValues* library (see 9.3.2), e.g., *Positive* for LiteralInfinity, *String* for LiteralString, etc.

Note. In the abstract syntax, the value property of LiteralRational has type Real (see 8.3.4.8.10), because that is the available UML/MOF primitive type. However, only the rational-number subset of the real numbers can be represented using a finite literal. So the result of a LiteralRational is actually always classified in the KerML DataType *Rational*.

8.4.4.9.3 Feature Reference Expressions

There is no specific specialization requirement for a FeatureReferenceExpression. However, the general checkExpressionSpecialization constraint (see 8.4.4.8) requires that a FeatureReferenceExpression specialize *Performances::Evaluation* (see 9.2.6.2.3). All other general semantic constraints for Expressions (see 8.4.4.8) also apply to FeatureReferenceExpressions.

A FeatureReferenceExpression is parsed with a non-owning Membership relationship to its referent Feature (see 8.2.5.8.3). The checkFeatureReferenceExpressionBindingConnector constraint then requires that there be a BindingConnector between this member Feature and the result parameter of the FeatureReferenceExpression. The checkFeatureResultSpecialization constraint further requires that the result parameter also subset the Feature. While this subsetting is technically implied by the semantics of the BindingConnector (see 8.4.4.6), including the Subsetting relationship allows for simpler static type checking of the result of the FeatureReferenceExpression.

Given the above, a FeatureReferenceExpression whose referent is a Feature *f* is semantically equivalent to the Expression

```
expr subsets Performances::evaluations {
    alias for f;
    return result
        redefines Performances::Evaluation::result
        subsets f;
    member binding result = f;
}
```

A body Expression (see 8.2.5.8.3) is parsed as a FeatureReferenceExpression that contains the Expression body as its *owned* referent. That is, a body Expression of the form

```
{ body }
```

is semantically equivalent to

```
expr subsets Performances::evaluations {
    expr e subsets Performances::evaluation { body }
    return result
        redefines Performances::Evaluation::result
        subsets e;
    binding result = e;
}
```

This means that the `result` of the `Expression` is the *`Evaluation`* of the body `Expression` itself, rather than the result of actually evaluating the body. If and when this *`Evaluation`* actually occurs can then be further constrained, e.g., within an invoked `Function` for which the body `Expression` is an argument (as done, for example, by *`ControlFunctions`* – see below).

8.4.4.9.4 Invocation Expressions

An `InvocationExpression` of the form *`F(e1, e2, ...)`*, where *`F`* is the name of a `Function` and *`e1, e2,`* ... are `argument Expressions`, is parsed with a `FeatureTyping` relationship to *`F`* and input `parameters` that have `FeatureValue` relationships to the `arguments` (see 8.2.5.8.3). There is no specific specialization requirement for an `InvocationExpression`. However, the general semantic constraints for `Expressions` (see 8.4.4.8) also apply to `InvocationExpressions`. Thus, an `InvocationExpression` of this form is semantically equivalent to

```
expr : F subsets Performances::evaluations {
    in a redefines F::a = e1;
    in b redefines F::b = e2;
    ...
    return result redefines F::result;
}
```

If the named-argument notation *`F(a = e1, b = e2, ...)`* is used, then the `InvocationExpression parameters` redefine the named `parameters` of *`F`*, regardless of order.

The semantic constraints for `FeatureValues` then require that each `parameter` is bound to the result of the corresponding `expression` (i.e., *`a`* is bound to *`e1.result`*, etc.). Thus, an `InvocationExpression` represents an `Evaluation` of the `Function` *`F`* with inputs corresponding to the results of evaluating the `argument Expressions`, producing result values in its `result` output `parameter`.

An `InvocationExpression` may also have the form *`T(e1, e2, ...)`*, where the invoked `Type` *`T`* is *not* a `Function`. If *`T`* is a `Behavior` other than a `Function`, then the `InvocationExpression` performs the `Behavior`, but has a **`null`** (empty) `result` value. If *`T`* is a `Type` that is not a `Behavior`, then the `InvocationExpression` acts as a *constructor* for an instance of the `Type` *`T`*. In this case the `checkInvocationExpressionBindingConnector` constraint requires that the `InvocationExpression` have an owned `BindingConnector` between itself and its `result` parameter (inherited from *`Performances::evaluations`*) – that is, the `InvocationExpression` evaluates, as an `Expression`, to itself, as an instance of *`T`*. Any `argument Expressions` are bound to `features` of the type *`T`* in the same way as describe above for `parameters`, with undirected `Features` treated as implicit inputs.

```
expr e : T subsets Performances::evaluations {
    in a redefines T::a = e1;
    b redefines T::b = e2;
    ...
    return result : T redefines Performances::evaluation::result;
    binding result = e;
}
```

Note that, in this case, the derived `function` of the `InvocationExpression` will always be the *`Performances::Evaluation`*, the type of *`Performances::evaluations`*.

If the invoked `Type` *`T`* names a `Feature`, or is a `Feature` chain, the semantics are similar, except that the `InvocationExpression` has a `Subsetting` relationship with *`T`*, instead of a `FeatureTyping` relationship. If the `Feature` is typed by a Function, then the `InvocationExpression` is effectively treated as an invocation of that `Function`. If the `Feature` is not typed by a `Function`, but is typed by a `Behavior`, then the `InvocationExpression` is treated as an invocation of that `Behavior` with a **`null`** result. Otherwise, the `InvocationExpression` is treated as a construction of an instance of the non-`Behavior` `type(s)` of the the `Feature`.

See also 8.4.4.11 on the semantic requirements for binding default `FeatureValues` in `InvocationExpressions`.

8.4.4.9.5 Operator Expressions

An `OperatorExpression` is an `InvocationExpression` in which the invoked `Function` is identified by an `operator` symbol. The `checkOperatorExpressionSpecialization` constraint requires that this `Function` be the resolution of the `operator` symbol as a name in one of the library `Packages` *`BaseFunctions`*, *`DataFunctions`* or *`ControlFunctions`*. The general semantic constraints for `Expressions` (see 8.4.4.9) also apply to `OperatorExpressions`.

With the exception of operators for *`ControlFunctions`* (see below), the concrete syntax for `OperatorExpressions` (see 8.2.5.8.1) is thus essentially just a special surface syntax for `InvocationExpressions` of the standard library `Functions` identified by their `operator` symbols. For example, a unary `OperatorExpression` such as

```
not expr
```

is equivalent to the `InvocationExpression`

```
DataFunctions::'not' (expr)
```

and a binary `OperatorExpression` such as

```
expr_1 + expr_2
```

is equivalent to the `InvocationExpression`

```
DataFunctions::'+' (expr_1, expr_2)
```

where these `InvocationExpressions` are then semantically interpreted as described 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.

Functions in the library models *`BaseFunctions`* and *`ScalarFunctions`* are extensively specialized in other library models to constrain their `parameter types` (e.g., the `Package` *`RealFunctions`* constrains `parameter types` to be *`Real`*, etc.). The result values the evaluation of such a `Function` shall be determined by the most specialized of its subtypes that is consistent with the `types` of its the dynamics result values from evaluating its `argument Expressions`.

Control Functions

Certain `OperatorExpressions` denote invocations of `Functions` in the *`ControlFunctions`* library model (see 9.4.17) that have one or more `parameters` that are `Expressions`. In the concrete syntax for such `OperatorExpressions` (see 8.2.5.8.1), the `arguments` corresponding to these `parameters` are parsed as if they were body `Expressions` (as described in 8.4.4.9.3), so they can effectively be passed without being immediately evaluated.

The second and third arguments of the ternary conditional test operator **`if`** are for `Expression` `parameters`. Therefore, the notation for a conditional test `OperatorExpression` of the form

```
if expr_1 ? expr_2 else expr_3
```

is parsed as

```
ControlFunctions::'if' (expr_1, { expr_2 }, { expr_3 })
```

The second arguments of the binary conditional logical operators **`and`**, **`or`**, and **`implies`** are for `Expression` `parameters`. Therefore, the notation for a conditional logical `OperatorExpression` of the form

```
expr_1 and expr_2
```

is parsed as

```
ControlFunctions::'and' (expr_1, { expr_2 })
```

and similarly for **`or`** and **`implies`**.

A `FeatureChainExpression` is an `OperatorExpression` whose operator corresponds to the `Function` *`ControlFunctions::'.'`*. This `Function` has a single `parameter` called *`source`*, but this `parameter` has a nested `Feature` called *`target`*. A `FeatureChainExpression` is parsed with an `argument` `Expression` for the *`source`* parameter and, additionally, a non-`parameter` `Membership` for its `targetFeature`, which is an alias `Membership` if the `targetFeature` is not a chain and an `OwningMembership` if the `targetFeature` is a chain. The `checkFeatureChainExpressionTargetRedefinition` constraint requires that the *`source`* `parameter` of the `FeatureChainExpression` have a nested `Feature` that redefines *`ControlFunctions::'.'::source::target`*, and the `checkFeatureChainExpressionSourceTargetRedefinition` requires that this nested `Feature` also redefine the `targetFeature`.

Given the above, a `FeatureChainExpression` of the form

```
src.f
```

(where *`src`* is an `Expression`) semantically equivalent to the `Expression`

```
expr : ControlFunctions::'.' subsets Performances::evaluations {
    feature redefines ControlFunctions::'.'::source = src {
        feature redefines ControlFunctions::'.'::source::target
            redefines f;
    }
    alias for f;
}
```

A `FeatureChainExpression` whose `targetFeature` is a `Feature` chain, of the form

```
src.f.g.h
```

is semantically equivalent to the `Expression`

```
expr : ControlFunctions::'.' subsets Performances::evaluations {
    feature redefines ControlFunctions::'.'::source = src {
        feature redefines ControlFunctions::'.'::source::target
            redefines tgt;
    }
    feature tgt chains f.g.h;
}
```

The performance of the `Function` `'.'` then results in the effective chaining of the value of its *`source`* parameter (which will be the result of the `argument` `Expression` of the `FeatureChainExpression`) and the *`source::target`* `Feature` (which will be the `targetFeature` of the `FeatureChainExpression`).

8.4.4.9.6 Metadata Access Expressions

The `checkMetadataAccessExpressionSpecialization` constraint requires that a `MetadataAccessExpression` specialize the `Expression` *`Performances::metadataAccessEvaluations`* (see 9.2.6.2.10), which is typed by the `Function` *`Performances::MetadataAccessEvaluation`* (see 9.2.6.2.9). The `result` `parameter` of *`MetadataAccessEvaluation`* is ordered and typed by *`Metaobjects::Metaobject`* (see 9.2.16.2.1). The general semantic constraints for `Expressions` (see 8.4.4.9) also apply to `MetadataAccessExpressions`.

A `MetadataAccessExpression` evaluates to an ordered set of *`Metaobjects`*, which are determined as follows:

- A *`Metaobject`* representing each `MetadataFeature` (see 8.3.4.12.3) owned by the `referencedElement` of the `MetadataAccessExpression` that has the `referenceElement` as an `annotatedElement`, in the order that the `MetadataFeatures` appear in the model. Each of these *`Metaobjects`* is an instance of the `metaclass` of the corresponding `MetadataFeature`, with the `features` of each instance having values determined by evaluating the bound `Expressions` of the `features` in the `MetadataFeature` as model-level evaluable `Expressions` (see below).
- Followed by a *`Metaobject`* that is an instance of the `Metaclass` from the reflective *`KerML`* abstract syntax library model (see 9.2.17) corresponding to the MOF metaclass of the `referencedElement` of the `MetadataAccessExpression`, with `features` having values corresponding to the values of the MOF properties for the `referencedElement`.

Note that every `Metaclass` is required to specialize *`Metaobjects::Metaobject`*, so the typing of the results of a `MetadataAccessExpression` is consistent.

For example, the `MetadataAccessExpression` *`C.`**`metadata`*** for the following `referencedElement`:

```
class C {
    metadata M;
}
```

would evaluate to two *`Metaobjects`*: an instance of the `Metaclass` *`M`* representing the `MetadataFeature` annotation on *`C`* and an instance of *`KerML::Class`* representing the `referencedElement` *`C`* itself.

8.4.4.9.7 Model-Level Evaluable Expressions

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. A model-level evaluable `Expression` is evaluated on a given *target* `Element` (see 8.4.4.13 and

8.4.4.14 for the targets used in the case of metadata `values` and `filterConditions`, respectively), using the `Expression::evaluate` operation, resulting in an ordered list of `Elements`. The rules for this operation are specified in the abstract syntax (see 8.3.4.8) and are summarized below:

1. A `NullExpression` evaluates to the empty list.
2. A `LiteralExpression` evaluates to itself.
3. A `FeatureReferenceExpression` is evaluated by first determining a value `Expression` for the `referent`:
   - If the `target Element` is a `Type` that has a `feature` that is the `referent` or (directly or indirectly) redefines it, then use the `value Expression` of the `FeatureValue` for that `feature` (if any).
   - Else, if the `referent` has no `featuringTypes`, then use the `value Expression` of the `FeatureValue` for the `referent` (if any).

   Then:
   - If such a value `Expression` exists, the `FeatureReferenceExpression` evaluates to the result of evaluating that `Expression` on the `target`.
   - Else, if the `referent` is not an `Expression`, the `FeatureReferenceExpression` evaluates to the `referent`.
   - Else, the `FeatureReferenceExpression` evaluates to the empty list.
4. A `MetadataAccessExpression` evaluates to the `ownedElements` of the `referencedFeature` that are `MetadataFeatures` and have the `referencedElement` as an `annotatedElement`, plus a `MetadataFeature` whose `annotatedElement` is the `referencedElement`, whose `metaclass` is the reflective `Metaclass` in the *KerML* library model (see 9.2.17) corresponding to the MOF class of the `referencedElement`, and whose `ownedFeatures` are bound to the values of the MOF properties of the `referencedElement`.
5. An `InvocationExpression` evaluates to an application of its `function` to argument values corresponding to 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` that is not an `Expression`. 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`.

8.4.4.10 Interactions Semantics

Abstract syntax reference: 8.3.4.9

8.4.4.10.1 Interactions

An `Interaction` is both an `Association` and a `Behavior`, and, therefore, the semantic constraints for both `Associations` (see 8.4.4.5) and `Behaviors` (see 8.4.4.7) apply. In particular, the `checkAssociationSpecialization` constraint requires that an `Interaction` specialize *`Links::Link`* (see 9.2.3.2.3), or, if it is a binary `Interaction` (with exactly two end `Features`), the `checkAssociationBinarySpecialization` constraint requires that it specializes *`Links::BinaryLink`* (see 9.2.3.2.1). And the `checkBehaviorSpecialization` constraint requires that it also specialize *`Performances::Performance`* (see 9.2.6.2.13).

These constraints require an N-ary `Interaction` to have the form (with implied relationships included)

```
interaction I specializes Link::Link, Performances::Performance {
    end feature e1 subsets Links::Link::participant;
```

```
        end feature e2 subsets Links::Link::participant;
        ...
        end feature eN subsets Links::Link::participant;
    }
```

with a binary `Interaction` having the form

```
    interaction B specializes Links::BinaryLink, Performances::Performance {
        end feature e1 redefines Links::BinaryLink::source;
        end feature e2 redefines Links::BinaryLink::target;
    }
```

The `checkFeatureEndRedefinition` and `checkFeatureParameterRefinition` constraints also apply to `Interactions`.

```
    interaction I1 specializes Links::BinaryLink, Performances::Performance {
        in feature x1;
        out feature y1;

        end feature e1;
        end feature f1;
    }
    interaction I2 specializes I1 {
        in feature x2 redefines x1;
        out feature y2 redefines y1;

        end feature e2 redefines e1;
        end feature f2 redefines f1;
    }
```

8.4.4.10.2 Item Flows

An `ItemFlow` is both a `Connector` and a `Step` and, therefore, the semantic constraints for both `Connectors` (see 8.4.4.6) and `Steps` (see 8.4.4.7) also apply to `ItemFlows`. In addition, the `checkItemFlowSpecialization` constraint requires that `ItemFlows` specialize *`Transfers::transfers`* (see 9.2.7.2.11). In addition, if the `ItemFlow` has `itemFlowEnds` (see below), then it must specialize *`Transfers::flowTransfers`* (see 9.2.7.2.4).

The textual notation for an `ItemFlow`, of the form

```
    flow of i : T from f1.f1_out to f2.f2_in;
```

is parsed with *`i : T`* as an `ItemFeature` and having two `ItemFlowEnds`, one referencing *`f1`* with an owned Feature redefining *`f1_out`* and one referencing *`f2`* with an owned Feature redefining *`f2_in`* (see 8.2.5.9.2). An `ItemFlowFeature` is just a `Feature` owned by an `ItemFlow` that has the special semantic constraint `checkItemFeatureRedefinition` that requires that an `ItemFeature` redefine *`Transfers::Transfer::item`* (see 9.2.7.2.9). An `ItemFlowEnd` is an end `Feature` owned by an `ItemFlow` that is required to have a single `ownedFeature`. The general `checkFeatureEndRedefinition` constraint (see 8.4.4.6) requires that the two `ItemFlowEnds` of an `ItemFlow` redefine *`Transfers::Transfer::source`* and *`Transfers::Transfer::target`* (see 9.2.7.2.9), respectively. The `checkFeatureItemFlowFeatureRedefinition` constraint then requires that the `ownedFeatures` of the `ItemFlowEnds` redefine *`Transfer::source::sourceOutput`* or *`Transfer::target::targetInput`*.

```
    flow subsets Transfers::flowTransfers {
        // ItemFeature
        feature i : T redefines Transfers::Transfer::item;
```

```
        // First ItemFlowEnd
        end feature redefines Transfers::Transfer::source references f1 {
            feature redefines Transfers::Transfer::source::sourceOutput, f1_out;
        }

        // Second ItemFlowEnd
        end feature references f2 redefines Transfers::Transfer::target {
            feature redefines Transfers::Transfer::target::targetInput, f2_in;
        }
    }
```

A `SuccessionItemFlow` is semantically the same, except that the `checkSuccessionItemFlowSpecialization` constraint requires that it specialize *`Transfers::flowTransfersBefore`* (see 9.2.7.2.3).

8.4.4.11 Feature Values Semantics

Abstract syntax reference: 8.3.4.10

A `FeatureValue` is a kind of `OwningMembership` between a `Feature` and an `Expression`. Note that the `FeatureValue` relationship is *not* a `Featuring` relationship, so its `featureWithValue` (that is, its owning `Feature`) is *not* the `featuringType` of the the value `Expression`. Instead, the `checkExpressionFeaturingType` constraint requires that the `value Expression` have the same `featuringTypes` as the `featureWithValue`. Most commonly, if the `featureWithValue` is an `ownedFeature` of a `Type`, this means that the `Expression` will have that `Type` as its `featuringType`.

The `checkFeatureValuationSpecialization` constraint requires that, if the `featureWithValue` has no explicit `ownedSpecializations` and is not directed, then it subsets the `result parameter` of the `value Expression`. This reflects the semantics that the values of the `featureWithValue` is determined by the `value Expression`, giving the featureWithValue an implied typing that is useful for static type checking. On the other hand, if the `featureWithValue` has `ownedSpecializations` or is directed, then its static typing can be considered determined by its declaration excluding the `FeatureValue` (but including any implied `Specializations`), which should then be validated against the typing of the `result` of the `value Expression`.

If the `FeatureValue` has `isDefault = false`, the `checkFeatureValueBindingConnector` constraint requires that its `featureWithValue` have an `ownedMember` that is a `BindingConnector` between that `Feature` and the `result parameter` of the value `Expression` of the `FeatureValue`. In addition, if the `FeatureValue` has `isInitial = false`, then the `featuringTypes` of this `BindingConnector` must be the same as those of the `featureWithValue`. Most commonly, if the `featureWithValue` is an `ownedFeature` of a `Type`, then the `BindingConnector` will have that `Type` as its `featuringType`. Other general semantic constraints for `Connectors` (see 8.4.4.6) also apply to the `BindingConnector` required for a `FeatureValue`.

Given the above, the textual notation for a `FeatureValue` with `isDefault = false` and `isInitial = false`, of the form

```
type T {
   feature f = expr;
}
```

is semantically equivalent to

```
type T {
    feature f {
        member expr e featured by T { ... }
        member binding f = e.result featured by T;
```

```
    }
}
```

where *e* is the semantic interpretation of *expr* as described in 8.4.4.9.

If a `FeatureValue` has `isDefault = false` but `isInitial = true`, then the `checkFeatureValueBindingConnector` constraint requires different `featuringTypes` for the `BindingConnector` than when `isInitial = false`. In this case, the `BindingConnector` must be featured by the *startShot* (see 9.2.4.2.13) of the *that* reference of its owning `featureWithValue` (see 9.2.2.2.7). Note that this is only possible if the `featureWithValue` is featured by a `Class` (see also 8.4.4.3 on the semantics of `Classes`). Most commonly, if the `featureWithValue` is an `ownedFeature` of a `Class` or a `Feature` typed by a `Class`, then the `BindingConnector` will have the `startShot` of that `Class` as its `featuringType`, meaning that the binding only applies initially, that is, at the very start of an *Occurrence* that is an instance of the `Class`.

Thus, the textual notation for a `FeatureValue` with `isDefault = false` and `isInitial = true`, of the form

```
class C {
   feature f := expr;
}
```

is semantically equivalent to

```
class C {
    feature f {
        member expr e featured by C { ... }
        member binding f = e.result featured by that.startShot;
    }
}
```

(note that the *that* is considered to be implicitly typed by *Occurrence* in this case).

If a `FeatureValue` has `isDefault = true`, then no `BindingConnector` is required for the `featureWithValue` at its point of declaration. Instead, the `checkInvocationExpressionDefaultValueBindingConnector` constraint requires that an `InvocationExpression` own a `BindingConnector` between the `featureWithValue` and value `Expression` of any `FeatureValue` that is the effective default value for a `Feature` of the invoked `Type` of the `InvocationExpression`, where *effective default value* is defined as follows:

- If the `Feature` has an owned `FeatureValue` with `isDefault = true`, then this is its effective default value.
- If the `Feature` does not have an owned `FeatureValue`, but the set of effective default values of the `Features` it redefines has a single unique member, then this is the effective default value of the original `Feature`.
- Otherwise the `Feature` does not have an effective default value.

For example, given the Type declaration

```
type T {
   feature f default = e;
}
```

a binding for `f` is included for the invocation *T()*, which is then semantically equivalent to

```
expr : T {
    binding f = f::e.result;
}
```

where *f::*`e.result` is the `result` of the `value Expression` from the default `FeatureValue`. On the other hand, for the invocation *`T(f = 1)`*, the `Feature` *`f`* will be bound to 1 rather than the `FeatureValue` default. A similar construction applies for `FeatureValues` with `isDefault = true` and `isInitial = true`. (See also 8.4.4.9 on the general semantics of `InvocationExpressions`.)

8.4.4.12 Multiplicities Semantics

Abstract syntax reference: 8.3.4.11

8.4.4.12.1 Multiplicities

A `Multiplicity` is a kind of Feature, so the general semantics of `Features` (see 8.4.3.4 also apply to a `Multiplicity`. In addition, the `checkMultiplicitySpecialization` constraint requires that a `Multiplicity` specialize the `Feature` *`Base::naturals`* (see 9.2.2.2.5), which is typed by the `DataType` *`ScalarValues::Natural`* (see 9.3.2.2.4). This constraint effectively requires that the co-domain of a `Multiplicity` be a subset of the natural numbers, which can be specified by reference to a library `Multiplicity` (such as *`Base:exactlyOne`* or *`Base::oneToMany`*) or using a `MultiplicityRange` from the Kernel layer (see 8.4.4.12.2).

The `validateTypeOwnedMultiplicity` constraint requires that a `Type` have at most one `ownedMember` that is a `Multiplicity`. If a `Type` has such an owned `Multiplicity`, then it is the `typeWithMultiplicity` of that `Multiplicity`. The value of the `Multiplicity` is then the *cardinality* of its `typeWithMultiplicity` and, therefore, the `type` (co-domain) of the `Multiplicity` restricts that cardinality. The cardinality of a `Type` is defined generally as follows:

- For a `Classifier`, the cardinality is the number of basic instances of the `Classifier`, that is, those instances that represent the things classified by the `Classifier` and are not instances of any subtype of the `Classifier` that is a `Feature`.
- For a `Feature`, the cardinality is the number of values of the `Feature` for any specific featuring instance (where duplicate features are included in the count, if the `Feature` is non-unique).

However, there are special rules for the semantics of `Multiplicity` for end `Features` (see 8.4.4.5).

The `checkMultiplicityTypeFeaturing` constraint requires that a `Multiplicity` with a `Feature` as its `owningNamespace` have the same `featuringTypes` (domain) as that `Feature`, and, otherwise, have no `featuringTypes`. In particular, a `Multiplicity` is owned by a `Feature` that has an `owningType`, then the `featuringType` of the `Multiplicity` is the `owningType` of its owning `Feature`. This means that the `Multiplicity` has a value for each instance of the `featuringType` that is the cardinality of the instances of its owning `Feature` that are featured by that same instance of the `featuringType`.

```
classifier C1 {
    feature f {
        // Implied TypeFeaturing by C2.
        // Gives the cardinality of the values of f for each
        // instance of C2 (which is constrained to be 1).
        multiplicity subsets Base::exactlyOne;
    }
}
```

If a `Type` does not have an owned `Multiplicity`, but has `ownedSpecializations`, then its cardinality is constrained by the `Multiplicities` for all of the general `Types` of those `ownedSpecializations` (i.e., its direct supertypes). In practice, this means that the effective `Multiplicity` of the `Type` is the most restrictive `Multiplicity` of its direct supertypes.

```
classifier C2 {
    feature f {
        multiplicity subsets Base::exactlyOne;
    }
    feature g {
        multiplicity subsets Base::oneToMany;
    }

    // The multiplicities exactlyOne and oneToMany both apply
    // to h, which means that, effectively, it has a multiplicity
    // of exactlyOne.
    feature h subsets f,g;
}
```

8.4.4.12.2 Multiplicity Ranges

A `MultiplicityRange` is a `Multiplicity` whose co-domain is given as an inclusive range of values of the type *`Natural`*. It thus constrains the cardinality of its `typeWithMultiplicity` to be within this range. A `MultiplicityRange` of the form

```
[expr_1.. expr_2]
```

represents the range of values 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`*. Note that all other *`Natural`* values are less than the value of `*`, representing positive infinity, so the `MultiplicityRange [0..*]` is the range of all values of *`Natural`* (that is, no restriction on cardinality).

A `MultiplicityRange` having only a single expression:

```
[expr]
```

is interpreted in one of the following ways:

- If *`expr`* evaluates to `*`, then it is equivalent to the range `[0..*]` (i.e., the entire extent of *Natural*).
- Otherwise, it is equivalent to `[expr..expr]` (that is, the cardinality is restricted to the single value given by the result of *`expr`*).

Note. The KerML textual notation grammar only allows `LiteralExpressions` and `FeatureReferenceExpressions` as the `bound Expressions` in a `MultiplicityRange` (see 8.2.5.11). However, the abstract syntax allows arbitrary `Expressions` (see 8.3.4.11).

The `checkMultiplicityRangeExpressionTypeFeaturing` constraint requirs that the `bound Expressions` of a `MultiplicityRange` have the same `featuringTypes` as the `MultiplicityRange`. The `featuringTypes` of a `MultiplicityRange` are determined by the `checkMultiplicityTypeFeaturing` constraint (8.4.4.12.1). If the `MultiplicityRange` has an `owningNamespace` that is not a `Feature`, then it has no `featuringTypes`, so its domain is implicitly *`Base::Anything`*, and its `bound Expressions` can only reference other `Features` in that context.

```
package P {
    // Implicitly featured by Anything.
```

```
        feature n : ScalarValues::Natural;
        classifier C3 {
            // An ownedMember, not an ownedFeature.
            // Implicitly featured by Anything.
            // Implied Subsetting of Base::naturals.
            multiplicity [P::n];
        }
    }
```

If the MultiplicityRange has an owningNamespace that is a Feature, then it is required to have featuringTypes that are the same as the owning Feature. In particular, if its owning Feature has an owningType, then the featuringType of the MultiplicityRange (and its bound Expressions) is the owningType of its owning Feature.

```
    classifier C4 {
        feature n : ScalarValues::Natural;
        feature m : Member {
            // Implied TypeFeaturing by C4.
            // Implied Subsetting of Base::naturals.
            multiplicity [1..C4::n];
        }
    }
```

8.4.4.13 Metadata Semantics

Abstact syntax reference: 8.3.4.12

8.4.4.13.1 Metaclasses

The checkMetaclassSpecialization constraint requires that Metaclasses specialize the base Metaclass *Metaobjects::Metaobject* (see 9.2.16.2.1). A Metaclass is a kind of Structure (see 8.4.4.4), but it's instances are *Metaobjects* that are part of the structure of a model itself, rather than as an instance in the system represented by the model. The *KerML* libary model is a reflective model of the MOF abstract syntax for KerML, containing one KerML Metaclass corresponding to each MOF metaclass in the abstract syntax model (see 9.2.17 for more details on the relationship between the *KerML* model and the abstract syntax).

8.4.4.13.2 Metadata Features

A MetadataFeature is both a Feature typed by a Metaclass and an AnnotatingElement that annotates other Elements in a model. The checkMetadataFeatureSpecialization requires that MetadataFeatures specialize the Feature *Metaobjects::metaobjects* (see 9.2.16.2.2). At a meta-level, a MetadataFeature can be treated as if the reflective Metaclasses of its annotatedElements were its featuringTypes. In this case, the MetadataFeature defines a map from its annotatedElements, as instances of their Metaclasses, to a single instance of the metaclass of the MetadataFeature.

Further, a model-level evaluable Expression is an Expression that can be evaluated using metadata available within a model itself (see 8.4.4.9). If a model-level evaluable Expression is evaluated on such metadata according to the regular semantics of Expressions, then the result will correspond to 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 value Expressions of its FeatureValues can be interpreted according to the regular semantics of FeatureValues (see 8.4.4.11) and BindingConnectors (see 8.4.4.6).

When a value Expression is model-level evaluated (as described in 8.4.4.9), its target is the MetadataFeature that owns the featureWithValue. 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.

8.4.4.13.3 Semantic Metadata

A *semantic* MetadataFeature is one that directly or indirectly specializes *Metaobjects::SemanticMetadata* (see 9.2.16.2.3) It is used to introduce a user-defined specialization constraint on the Type annotated by the MetadataFeature. *SemanticMetadata* has the Feature *baseType* typed by the reflective Metaclass *KerML::Type* (see 9.2.17) that is redefined by a semantic MetadataFeature. The target of the effective specialization constraint defined by a semantic MetadataFeature is determined by the value Expression bound to its *baseType* Feature using a FeatureValue (see 8.4.4.11), which is evaluated as a model-level evaluable Expression (see 8.4.4.9).

Specifically, for each semantic MetadataFeature annotating a Type, the checkMetadataFeatureSemanticSpecialization constraint requires that the annotated Type directly or indirectly specialize the Type bound to the *baseType* of the MetadataFeature, *unless* the annotated Type is a Classifier and the *baseType* is a Feature. For the case when the Type is a Classifier and the *baseType* is a Feature, the constraint requires that the annotated Classifier directly or indirectly specialize each type of the *baseType* Feature.

8.4.4.14 Packages Semantics

Abstract syntax reference: 8.3.4.13

Packages do not have instance-level semantics (they do not affect instances).

The filterConditions of a Package are model-level evaluable Expressions that are evaluated as described in 8.4.4.9. All filterConditions are checked against every Membership that would otherwise be imported into the Package if it had no filterConditions. 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 MetadataFeature of the memberElement of the Membership as the target Element.

9 Model Libraries

9.1 Model Libraries Overview

A *model library* is a collection of library models that can be reused across many user models. KerML includes three standard model libraries: the Semantic Library (see 9.2), the Data Type Library (see 9.3), and the Function Library (see 9.4). Each library model in these standard model libraries consists of a single root namespace with one top-level element that is a standard library package, with no subpackages. All of these library models are described for reference in subclauses of this clause.

The normative machine-readable representation for each of these model libraries is a project interchange file, formatted according to the standard for KerML model interchange given in Clause 10. Each library model is packaged as a model interchange file in the project interchange file for its corresponding model library. Regardless of whether such a library model is interchanged in textual notation, XMI or JSON format, the `elementId` for any `Element` in the library model that has a non-null `qualifiedName` shall be a name-based (version 5, using SHA-1) UUID (see [UUID, 14.3]), which are constructed from a *name space identifier* and a *name* determined as follows:

- For the top-level standard library package:
  - *name space identifier* shall be the `NameSpace_URL` UUID, as given in [UUID, D.9] (which is `6ba7b812-9dad-11d1-80b4-00c04fd430c8`).
  - *name* shall be the URL constructed by prepending `https://www.omg.org/KerML/` to the name of the package, converted to bytes using a `UTF-8` encoding (see [ISO10646, Annex D]).
- For any element directly or indirectly contained in the top-level standard library package (for which that package will be the `libraryNamespace`):
  - *name space identifier* shall be the UUID of the top-level standard library package (as determined above).
  - *name* shall be the `qualifiedName` of the element , converted to bytes using a `UTF-8` encoding (see [ISO10646, Annex D]).

The `elementIds` constructed as given above shall be normative across all forms of interchange of the library models and shall remain stable for future versions of the library models, though future revisions of this specification may deprecate certain existing `Elements` and their names, or introduce new `Elements` with new names and hence UUIDs that are distinct (with a high probability). However, the `elementIds` for library `Elements` for which the `qualifiedName` is null are *not* determined by this specification, and representations of the models with different `elementIds` for these `Elements` (such as may happen when parsing the textual representation of a library model) are still considered conformant to the specification.

9.2 Semantic Library

9.2.1 Semantic Library Overview

The Semantic Library is a collection of KerML models that are part of the semantics of the metamodel (see Clause 8). They are reused when constructing KerML user models (instantiating the abstract syntax), as specified by constraints and semantics of metaelements, such as Types being required to specialize *`Anything`* from the library and Behaviors specializing *`Performance`* (see 8.4). The library can be specialized for particular applications, such as systems modeling.

The Semantic Library contains a set of packages, one for each library model, as described in a subsequent subclauses. The following are the major areas covered in the Semantic Library.

1. The *`Base`* library model (see 9.2.2) 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 9.2.3) specializes *Base* to provide the semantics for Associations between things.

2. The *Occurrences* library model (see 9.2.4) 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 9.2.5) specializes *Occurrences* to provide a model of *Objects* and *LinkObjects*, giving semantics to Structures and AssociationStructures, respectively. The *Performances* library model (see 9.2.6) 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 9.2.7) models asynchronous flow of items between *Occurrences,* giving semantics to Interactions and ItemFlows. The *FeatureAccessPerformances* library model (see 9.2.8) 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 9.2.9, 9.2.10, and 9.2.11) 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 using these library models, can add syntax as needed by their applications.

9.2.2 Base

9.2.2.1 Base Overview

This library model begins the Specialization hierarchy for all KerML Types (see 8.3.3.1 and 8.4.3.2), 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 8.3.4.1). *DataValues* are *Anything* that can only be distinguished by how they are related to other things (via Features and Assocations). These 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 9.3.2.1). The Feature *self* of *Anything* relates each thing in the universe to itself only (see *SelfLinks* in 9.2.3.1).

9.2.2.2 Elements

9.2.2.2.1 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

self : Anything {subsets selfSameLife}

The source of a SelfLink of this thing to itself. `self` is thus a feature that relates everything to itself. It is also the value of the nested `that` feature of all other things featured by this thing.

Constraints

None.

9.2.2.2.2 DataValue

Element

DataType

Description

A DataValue is Anything that can only be distinguished by how it is related 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.

9.2.2.2.3 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.

9.2.2.2.4 exactlyOne

Element

MultiplicityRange

Description

`exactlyOne` is a `MultiplicityRange` requiring a cardinality of exactly one.

General Types

naturals

Features

None.

Constraints

None.

9.2.2.2.5 naturals

Element

Feature

Description

General Types

Natural
dataValues

Features

None.

Constraints

None.

9.2.2.2.6 oneToMany

Element

MultiplicityRange

Description

oneToMany is a `MultiplicityRange` requiring a cardinality of one or more.

General Types

naturals

Features

None.

Constraints

None.

9.2.2.2.7 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.

`things` has multiplicity lower bound 1 because, for any featuring instance, it includes at least that instance as the value of `Anything::self`.

General Types

Anything

Features

that : Anything

For each value of `things`, the "featuring instance" of that value. Formally, for any sequence *s* classified by `things`, the `that` includes a sequence whose prefix is *s*, followed by the second-to-last element of *s*. This is enforced by declaring `Anything::self` to be the chaining of `things.that`, restricting `that` to the single value of `self` for all `things`.

Constraints

None.

9.2.2.2.8 zeroOrOne

Element

MultiplicityRange

Description

zeroOrOne is a `MultiplicityRange` requiring a cardinality of zero or one.

General Types

naturals

Features

None.

Constraints

None.

9.2.2.2.9 zeroToMany

Element

MultiplicityRange

Description

`zeroToMany` is a `MultiplicityRange` requiring a cardinality of zero or more.

General Types

naturals

Features

None.

Constraints

None.

9.2.3 Links

9.2.3.1 Links Overview

This library model introduces the most general Association *Link*, the type of *`links`*, the most general Feature typed by Associations (see 8.3.4.4 and 8.4.4.5). 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 7.4.6.3). *SelfLinks* are not in time or space (they are not Occurrences, see 9.2.4).

9.2.3.2 Elements

9.2.3.2.1 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.

9.2.3.2.2 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.

9.2.3.2.3 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.

9.2.3.2.4 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.

9.2.3.2.5 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

sameThing : Anything {subsets thisThing, redefines source}

The source participant of this SelfLink, which must be the same as the target participant.

self2 : Anything {subsets myselfSameLife}

The target end of a SelfLink.

thisThing : Anything {subsets sameThing, redefines target}

The target participant of this SelfLink, which must be the same as the source participant.

Constraints

None.

9.2.3.2.6 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.

9.2.4 Occurrences

9.2.4.1 Occurrences Overview

Occurrences

This library adds a model of things existing in time and space, starting with `Occurrence`, the most general Class (see 8.3.4.2), which classifies *Anything* that takes up time and space, and `occurrences`, the most general Feature typed by Classes. *Occurrences* can take up the same or overlapping time and space when they represent different things happening or existing in it. For example, the time and space taken by a room might have air moving in it it, as well as light, radio waves, and so on.

Occurrences divide into *Objects* and *Performances* (see 9.2.5.1 and 9.2.6.1, respectively), corresponding to Classes dividing into Structures and Behaviors (see 7.4.4 and 7.4.7.1, respectively). This subclause covers what is in common between *Objects* and *Performances*.

Temporal and Spatial Associations

Occurrences can be completely separated in time or space, or both, as indicated by these specialized *Links*:

- *HappensBefore Links* between *Occurrences* indicate they are completely separate in time, with one happening or existing completely before another. 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 or existing 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. *Occurrences* separated in time are not necessarily separated in space.

- *OutsideOf Links* between *Occurrences* indicate they are completely separate in space, without specifying their relative positions (such as above or to the left). The `outsideOccurrences` of *Occurrences* are those that exist *OutsideOf* them. *JustOutsideOf Links* are *OutsideOf Links* between *Occurrences* where there is no possibility of other *Occurrences* happening or existing in the space between at least some of their `spaceBoundaries`, see space boundaries below. The `justOutsideOccurrences` of *Occurrences* are those that exist *JustOutsideOf* them. *Occurrences* separated in space are not necessarily separated in time.

Without Links between *Occurrences* are provided as a convenience to indicate one *HappenBefore* another or is *OutsideOf* the other or both (they do not overlap at all in space-time), with the `withoutOccurrences` of an *Occurrence* being the ones that are *Without* it.

Occurrences can completely overlap others in time or space, or both, as indicated by these specialized *Links*:

- *HappensDuring Links* between *Occurrences* indicate one happens or exists completely within the time taken by another, with the `timeEnclosedOccurrences` of an *Occurrence* being the ones that *HappenDuring* it. *Occurrences* overlapping in time do not necessarily overlap in space.

- *InsideOf Links* between *Occurrences* indicate one happens or exists completely within the space taken by another, with the `spaceEnclosedOccurrences` of an *Occurrence* being the ones that *InsideOf* it. *Occurrences* overlapping in space do not necessarily overlap in time.

Within Links between *Occurrences* are provided as a convenience to indicate one *HappensDuring* another and is *InsideOf* that other (one is completely overlapped by the other in space-time), with the `spaceTimeEnclosedOccurrences` of an *Occurrence* being the ones that are *WithIn* it.

Occurrences cannot be linked by both *HappensBefore* and *HappensDuring*, *OutsideOf* and *InsideOf*, or *Within* and *Without*. They also cannot *HappenBefore* or be *OutsideOf* or *Without* themselves, but always *HappenDuring* and are *InsideOf* and *Within* themselves. When an *Occurrence HappensBefore* another, all *Occurrences* that *HappenDuring* the earlier one (including itself) also *HappenBefore* those that *HappenDuring* the later one (including itself).

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*, with the `timeCoincidentOccurrences` of an *Occurrence* being the ones that *HappenWhile* it. *Occurrences* that are *InsideOf* each other both ways occupy exactly the same space, even though they might happen or exist at separate times. *Occurrences* that are *Within* each other both ways happen at exactly the same time and occupy exactly the same space, which is provided for convenience by *WithinBoth*, a specialization of *Within*, with the `spaceTimeCoincidentOccurrences` of an *Occurrence* being the ones that are *WithinBoth* it.

The *Links* above to do not take up time or space, they are temporal and spatial relations between things that do (they are disjoint with *LinkObject*, see 9.2.5.1).

Other Time-Space Relations

The time and space taken by an *Occurrence* can be related in three ways to the time and space taken by others, identified by the Features below. An *Occurrence* with values for these Features takes the same time and space as

- `unionOf`: taken by all the other *Occurrences* together.
- `intersectionOf`: is common to all the other *Occurrences*.
- `differencesOf`: the first other *Occurrence* that is not taken by the rest.

The values of the above Features are *Sets* of *Occurrences* to enable the time and space of an *Occurrence* to be specified in multiple ways, with each set taken as a complete specification of the time and space taken by the *Occurrence*.

Portions

It is useful to consider *Occurrences* during only some of the time and space they take up, which are other *Occurrences* identified as `portions` (the most general portion Feature, see 9.2.4.2.13 and 8.3.3.3). These are the same "thing" as their larger *Occurrences*, just considered for a potentially smaller period of time and region in space. They must be classified the same way as the *Occurrences* they are `portionsOf`, or more specialized.

Occurrences are always `portionsOf` themselves. *Occurrences* that are only `portionsOf` of themselves are *Lives* (classified 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`. This means following `portionsOf` repeatedly will always reach a single *Life*, even though some *Occurrences* along the way might be `portionsOf` of more than one other *Occurrence*.

SelfSameLifeLinks include *SelfLinks* (*Links* between each thing and itself, see 9.2.3.1), as well as *Links* between *Occurrences* that are `portionsOf` the same *Life* (have the same `portionOfLife`).

Time and Space Slices

Time slices are `portions` that include all the space of their larger *Occurrences* within a potentially smaller period of time than the whole *Occurrence*, identified as `timeSlices` of the *Occurrences* they are `portionsOf`. Time slices might have Feature values and *Links* to other things peculiar to their smaller period of time. *Occurrences* are always `timeSlicesOf` themselves. The `snapShots` of *Occurrences* are `timeSlices` that take no time. The earliest `snapShot` of an *Occurrence* is its `startShot`, the latest is its `endShot`. All the others happen during its `middleTimeSlice`. *Occurrences* with a `startShot` the same as their `endShot` take no time, have no `middleTimeSlice`, and vice-versa.

Space slices are `portions` that include all the space of their larger *Occurrences*, but not necessary all their time, identified as `spaceSlices` of the *Occurrences* they are `portionsOf`. Space slices might have Feature values and Links to other things peculiar to their smaller region in space. *Occurrences* are always `spaceSlicesOf` themselves. The `spaceShots` of *Occurrences* are `spaceSlices` that have a lower `innerSpaceDimension` than the *Occurrences* they are `spaceSlicesOf`, which is the number of variables needed to identify any space point occupied by an *Occurrence*, without regard to higher dimensional spaces in which it might be embedded. For example, the `innerSpaceDimension` of a *Curve* is 1 (see 9.2.5.1), because points on it can be identified by the distance from one end, even if the curve bends in two or three dimensions. A *Curve* can be a `spaceShot` of a *Surface* or *Body*, which have `innerSpaceDimension` of 2 and 3, respectively. The `spaceSlices` of an *Occurrence* that are not `spaceShots` must have the same `innerSpaceDimension` as the *Occurrence*. How much an *Occurrence* bends in higher dimensions is its `outerSpaceDimension` (see 9.2.5.1). For example, the `outerSpaceDimension` of a planar curve is 2 or 1 (*Line*), while it is 3 for non-planar.

Space Boundaries and Interiors

The `spaceSlices` of each *Occurrence* are divided into a `spaceBoundary`, which is a `spaceShot`, and a `spaceInterior`, which is a `spaceSlice` that is not a `spaceShot` (has the same `innerSpaceDimension` as the *Occurrence*). They are *JustOutsideOf* each other and union (see below) to the entire *Occurrence*. Space boundaries cannot have a `spaceBoundary`, which means they also cannot have a `spaceInterior`, indicated by `isClosed`=true, For example, a ball has a sphere as its `spaceBoundary`, but the sphere `isClosed`.

A `spaceBoundary` might have `spaceSlices` that are also closed and have the same `innerSpaceDimension` as the `spaceBoundary` (not among its `spaceShots`). In some cases one of these `spaceSlices` surrounds the others, identified as the `outer`, a nested feature of `spaceBoundary`, and the others as the `inner` ones. This means the `outer` one can be taken as the `spaceBoundary` of another *Occurrence* with a `spaceInterior` that completely includes the `inners`. The `inner spaceBoundaries` can also be taken as `spaceBoundaries` of their own *Occurrences*, the `spaceInteriors` of which are identified as the `innerSpaceOccurrences` ("holes") of the *Occurrence* having the `spaceBoundary`. These two cases are covered by *SurroundedBy Links* between *Occurrences*, with the `surroundedByOccurrences` of an *Occurrence* being the ones they are *SurroundedBy*.

MatesWith Links are *JustOutsideOf Links* between *Occurrences* indicating that they union (see below) to an *Occurrence* with a `spaceBoundary` but no `spaceInterior`. This means there is no possibility of other *Occurrences* happening or existing in the space between them. *JustOutsideOf Links* additionally include those between *Occurrences* where only some of their `spaceSlices` (of their `spaceBoundaries`) are linked by *MatesWith*.

9.2.4.2 Elements

9.2.4.2.1 HappensBefore

Element

Association

Description

HappensBefore is a Without association 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 {subsets sourceOccurrence, redefines separateOccurrenceToo}

The participant of this HappensBefore link that happens (ends) earlier than the other participant (starts).

laterOccurrence : Occurrence {subsets targetOccurrence, redefines separateOccurrence}

The participant of this HappensBefore link that happens later than (starts after) the other participant (ends).

Constraints

None.

9.2.4.2.2 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.

9.2.4.2.3 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

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.

longerOccurrence : Occurrence {redefines targetOccurrence}

The participant in this HappensDuring Link that completely overlaps the other in time.

shorterOccurrence : Occurrence {redefines sourceOccurrence}

The participant in this HappensDuring Link that is completely overlapped by the other in time.

Constraints

None.

9.2.4.2.4 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

earlierOccurrence : Occurrence {redefines earlierOccurrence}

laterOccurrence : Occurrence {redefines laterOccurrence}

Constraints

None.

9.2.4.2.5 HappensLink

Element

Association

Description

HappensLink is the most general association that asserts temporal relationships between a `sourceOccurrence` and a `targetOccurrence`. They cannot happen in time (be Occurrences), making them disjoint with LinkObject.

General Types

BinaryLink

Features

happensSource : Occurrence [0..*] {subsets toSources}

happensTarget : Occurrence [0..*] {subsets toTargets}

sourceOccurrence : Occurrence {redefines source}

targetOccurrence : Occurrence {redefines target}

Constraints

None.

9.2.4.2.6 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}

timeCoincidentOccurrences : Occurrence [1..*] {subsets happensDuring}

Occurrences that start and end at the same time as this one.

Constraints

None.

9.2.4.2.7 IncomingTransferSort

Element

Predicate

Description

A predicate of two transfers that is true when the first should be accepted instead of the other.

General Types

BooleanEvaluation

Features

None.

Constraints

None.

9.2.4.2.8 InnerSpaceOf

Element

Association

Description

InnerSpaceOf is an OutsideOf asserting that the space surrounded by an inner space boundary of one occurrence (`outerSpace`) is completely occupied by another occurrence (`innerSpace`).

General Types

OutsideOf

Features

innerSpace : Occurrence {redefines separateSpace}

The participant of this InnerSpaceOf link that completely occupy the space surrounded by an inner space boundary of the other.

innerSpaceOccurrenceOf : Occurrence [0..*] {subsets outsideOfOccurrences}

outerSpace : Occurrence {redefines separateSpaceToo}

The participant of this InnerSpaceOf link with an inner space boundary is completely occupied by the other.

Constraints

None.

9.2.4.2.9 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

insideOf : Occurrence [1..*] {subsets toTargets}

Occurrences that completely overlap this one in space (not necessarily in time, see `happensDuring`), including this one.

largerSpace : Occurrence {redefines target}

The participant in this InsideOf Link that completely overlaps the other in space.

smallerSpace : Occurrence {redefines source}

The participant in this InsideOf Link that is completely overlapped by the other in space.

Constraints

None.

9.2.4.2.10 JustOutsideOf

Element

Association

Description

JustOutsideOf is an OutsideOf association linking two occurrences have some space slices with no space between them.

General Types

OutsideOf

Features

justOutsideOfOccurrences : Occurrence [0..*] {subsets outsideOfOccurrences}

Occurrences that have no space between some of their space slices and some space slices of this occurrence.

separateSpace : Occurrence {redefines separateSpace}

separateSpaceToo : Occurrence {redefines separateSpaceToo}

Constraints

None.

9.2.4.2.11 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.

9.2.4.2.12 MatesWith

Element

Description

General Types

None.

Features

matesWith : Occurrence [0..*] {subsets justOutsideOfOccurrences}

Constraints

None.

9.2.4.2.13 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).

dispatchScope : 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 snapshots}

The `snapshot` of this Occurrence that `happensAfter` all its other `snapshots`.

immediatePredecessors : Occurrence [0..*] {subsets predecessors}

Occurrences that start just after this occurrence ends, with no possibility of other occurrences happening in the time between them.

immediateSuccessors : Occurrence [0..*] {subsets successors}

Occurrences that end just before this occurrence starts, with no possibility of other occurrences happening in the time between them.

incomingTransfer : Transfer [0..*]

incomingTransferSort : IncomingTransferSort [0..*]

Determines which transfers to accept when multiple are available and which of the unaccepted transfers are never to be accepted (dispatched), by comparing two transfers at a time. Defaults to `earlierFirstIncomingTransferSort`, which is true if the first transfer ends (arrives) before the other.

incomingTransferToSelf : Transfer [0..*] {subsets incomingTransfer}

Transfers for which this Occurrence is the `targetParticipant`.

inner : Occurrence [0..*]

A `spaceSlice` of `spaceBoundary`, see `spaceBoundary`.

innerSpaceDimension : Natural

The number of variables needed 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`.

innerSpaceOccurrences : Occurrence [0..*] {subsets outsideOfOccurrences}

Occurrences that completely occupy the space surrounded by an inner space boundary of this occurrence.

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.

isDispatch : Boolean

Determines whether the same incoming transfer can be accepted more than once by StatePerformances composed under `dispatchScope`. It defaults to `true` for Performances, and `false` for other Occurrences (including Objects).

isRunToCompletion : Boolean

Determines whether TransitionPerformances composed under `runToCompletionScope` can happen during StatePerformance `entry` Performances composed under this Occurrence.

justOutsideOfOccurrences : Occurrence [0..*] {subsets outsideOfOccurrences}

Occurrences that have no space between some of their space slices and some space slices of this occurrence.

localClock : Clock

A local *`Clock`* to be used as the corresponding time reference for this *`Occurrence`* and, by default, all *`ownedOccurrences`*. By default this is the singleton *`Clocks::universalClock`*.

matingOccurrences : Occurrence [0..*] {subsets justOutsideOfOccurrences}

Occurrences that have no space between them and this one.

middleTimeSlice : Occurrence [0..1] {subsets timeSlices}

`timeSlice` of this Occurrence that takes all of the time between its `startShot` and `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.

minuend : Occurrence [0..1] {subsets }

A (nested) feature of `differencesOf` that identifies the first Occurrence in its `elements`.

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.

outer : Occurrence [0..1]

A `spaceSlice` of `spaceBoundary`, see `spaceBoundary`.

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`.

outsideOfOccurrences : Occurrence [0..*] {subsets withoutOccurrences}

Occurrences that are completely separate from this one in space (not necessarily in time, see `successors` and `predecessors`).

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`.

portions : Occurrence [1..*] {subsets spaceTimeEnclosedOccurrences}

All occurrences `within` this one that are considered the same thing occurring (same `portionOfLife`), including this one.

predecessors : Occurrence [0..*] {subsets withoutOccurrences}

Occurrences that are completely separate from this one in time (not necessarily in space, see `outsideOfOccurrences`) and that happen before this one (end earlier than this one starts).

runToCompletionScope : Occurrence

sameLifeOccurrences : Occurrence [1..*]

self : Occurrence {subsets timeSlices, spaceSlices, spaceTimeCoincidentOccurrences, sameLifeOccurrences, redefines self}

This Occurrence (related to itself via a SelfLink).

snapshotOf : Occurrence [0..*] {subsets timeSliceOf}

Occurrences of which this Occurrence is a `snapshot`.

snapshots : Occurrence [1..*] {subsets timeSlices}

All `timeSlices` of this Occurrence that happen at a single instant of time (zero duration).

spaceBoundary : Occurrence [0..1] {subsets spaceShots}

A `spaceShot` of this Occurrence that is not among those of its `spaceInterior`, which it must be OutsideOf. It must not have a `spaceBoundary` (`isClosed` = true). It can be divided into `spaceSlices` that also have no `spaceBoundary`, where the `inner` ones are `SurroundedBy` the `outer` one.

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`.

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 portions}

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 timeCoincidentOccurrences, spaceEnclosedOccurrences}

Occurrences that this one completely includes in both space and time, and vice-versa, including this one.

spaceTimeEnclosedOccurrences : Occurrence [1..*] {subsets spaceEnclosedOccurrences, timeEnclosedOccurrences}

All `timeEnclosedOccurrences` of this one that are also `spaceEnclosedOccurrences`, 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 snapshots}

The `snapshot` of this Occurrence that `happensBefore` all its other `snapshots`.

suboccurrences : Occurrence [0..*]

Composite *`suboccurrences`* of this *`Occurrence`*. The *`localClock`* of all *`suboccurrences`* defaults to the *`localClock`* of its containing *`Occurrence`*.

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 withoutOccurrences}

Occurrences that are completely separate from this one in time (not necessarily in space, see `outsideOfOccurrences`) and that happen after this one (start later than this one ends).

surroundedByOccurrences : Occurrence [0..*] {subsets outsideOfOccurrences}

Occurrences that have inner spaces that completely include this occurrence.

this : Occurrence

The "context" Occurrence within which this Occurrence takes place. By default, it is this Occurrence itself. However, this is overridden for `ownedPerformances` of Objects and `subperformances` of Performances.

timeCoincidentOccurrences : Occurrence [1..*] {subsets timeEnclosedOccurrences}

Occurrences that `happenWhile` this one does (Occurrences that start and end at the same time as this one).

timeEnclosedOccurrences : Occurrence [1..*] {subsets happensSource}

Occurrences that this one completely overlaps in time (not necessarily in space, see `spaceEnclosedOccurrences`; 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 portions}

`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).

withoutOccurrences : Occurrence [0..*] {subsets toTargets}

All Occurrences that are `successors`, `successorsoutsideOf` of this one.

Constraints

None.

9.2.4.2.14 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.

9.2.4.2.15 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

outsideOfOccurrences : Occurrence [0..*] {subsets withoutOccurrences}

Occurrences that are completely separate from this one in space (not necessarily in time, see `successors` and `predecessors`).

separateSpace : Occurrence {redefines separateOccurrence}

The second participant in this OutsideOf Link.

separateSpaceToo : Occurrence {redefines separateOccurrenceToo}

The first participant in this OutsideOf Link.

Constraints

None.

9.2.4.2.16 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.

9.2.4.2.17 SelfSameLifeLink

Element

Association

Description

SelfLifeLinks are all the BinaryLinks such that 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

myselfSameLife : Anything [1..*] {redefines toSources}

The target end of a SelfLifeLink.

selfSameLife : Anything [1..*] {redefines toTargets}

The source end of a SelfLifeLink.

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.

9.2.4.2.18 SnapshotOf

Element

Association

Description

SnapshotOfis 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.

9.2.4.2.19 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`.

spaceShotOf : Occurrence [1..*] {subsets spaceSliceOf}

All `spaceSlicesOf` this Occurrence that are of a higher `innerSpaceDimension` than this Occurrence.

spaceShottedOccurrence : Occurrence {redefines spaceSlicedOccurrence}

The participant in this SpaceShotOf Link that is the `spaceSliced Occurrence`.

Constraints

None.

9.2.4.2.20 SpaceSliceOf

Element

Association

Description

SpaceSliceOfis 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.

9.2.4.2.21 SurroundedBy

Element

Association

Description

SurroundedBy is an OutsideOf asserting that one occurrence (`surroundedSpace`) is included in space by an `iinnerSpaceOccurrence` of another (`surroundingSpace`).

General Types

OutsideOf

Features

surroundedSpace : Occurrence {redefines separateSpaceToo}

The participant of this SurroundedBy link that is completely included in the an inner space of the other.

surroundingSpace : Occurrence {redefines separateSpace}

The participant of this SurroundedBy link that has an inner space that completely includes the other.

surroundsOccurrences : Occurrence [0..*] {subsets outsideOfOccurrences}

Constraints

None.

9.2.4.2.22 TimeSliceOf

Element

Association

Description

TimeSliceOfis 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.

9.2.4.2.23 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`.

within : Occurrence [1..*] {subsets insideOf, happensDuring}

All Occurrences that this one `happensDuring` and is `insideOf`, including this one.

Constraints

None.

9.2.4.2.24 WithinBoth

Element

Association

Description

WithinBoth 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 WithinBoth with itself and WithinBoth is transitive.

General Types

HappensWhile
Within

Features

spaceTimeCoincidentOccurrences : Occurrence [1..*] {subsets timeCoincidentOccurrences, within}

Occurrences that this one completely includes in both space and time, including this one.

thatOccurrence : Occurrence {redefines largerOccurrence}

thisOccurrence : Occurrence {redefines smallerOccurrence}

Constraints

None.

9.2.4.2.25 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 and/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.

withoutOccurrences : Occurrence [0..*] {subsets toSources}

All Occurrences that are `successors`, `successorsoutsideOf` of this one.

Constraints

None.

9.2.5 Objects

9.2.5.1 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 8.3.4.3 and compare to *Performances* in 9.2.6.1). *Objects* and *Performances* do not overlap, but *Performances* can Involve *Objects*, which can *Perform Performances*.

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*.

Body(s), *Surfaces*, *Curves*, and *Points* are *Objects* with `innerSpaceDimension` of 3, 2, 1, and 0, respectively.

Structured Space Objects

StructuredSpaceObjects are *Objects* with three Features Subsetting `spaceSlices`:

- `faces`, identifying *Surfaces*.
- `edges`, identifying *Curves*.
- `vertices`, identifying *Points*.

The above are collectively `structuredSpaceCells`, which are also *StructuredSpaceObjects*, enabling *faces* to identify `edges` and `vertices` among the `spaceSlices` of their `spaceBoundaries`, if any, and `edges` to identify `vertices` among theirs. Cells of closed *StructuredSpaceObjects* (`isClosed = true`) must be *JustOutside* others along their entire `spaceBoundary` (every cell's `spaceSlices` must *MateWith* some `spaceSlice` of another cell, see Space Boundaries and Interiors in 9.2.4.1), which usually means all the `edges` and `vertices` of cells *MateWith* those of other cells, enabling the *StructuredSpaceObject* to be the `spaceBoundary` for other *Objects*. The `innerSpaceDimension` of a *StructuredSpaceObject* is the highest `innerSpaceDimension` of its `structuredSpaceCells`.

Models can specialize the three Features above for various kinds of *Objects*, for example, one for cylinders would include:

- Three Features Subsetting `faces` for the top, bottom, and middle *Surfaces* of a cylinder. The `edges` of these Features are *Curves* (circles) that are `spaceBoundaries` of the top and bottom *Surfaces* (discs), and `spaceSlices` of the `spaceBoundary` of the middle *Surface* (a rectangle joined at two opposite sides).
- Two Features Subsetting `edges` for the top and bottom of the cylinder. Each Feature identifies two *Curves* that are the *edges* of adjacent *faces*, specified by *BindingConnectors* between the Feature and required `edges`. These two *Curves* must mate, specified by a `MateWith` Connector between the Feature and itself.
- A Feature redefining `vertices` to multiplicity 0.

9.2.5.2 Elements

9.2.5.2.1 BinaryLinkObject

Element

AssociationStructure

Description

General Types

LinkObject
BinaryLink

Features

toSources : Anything [0..*] {redefines toSources}

toTargets : Anything [0..*] {redefines toTargets}

Constraints

None.

9.2.5.2.2 binaryLinkObjects

Element

Feature

Description

General Types

linkObjects
BinaryLinkObject
binaryLinks

Features

[no name] : Anything

[no name] : Anything

Constraints

None.

9.2.5.2.3 Body

Element

Structure

Description

Objects of `innerSpaceDimension` 3.

General Types

Object

Features

innerSpaceDimension : Integer {redefines innerSpaceDimension}

volume

Constraints

None.

9.2.5.2.4 Curve

Element

Structure

Description

Objects of `innerSpaceDimension` 1.

General Types

Object

Features

innerSpaceDimension : Integer {redefines innerSpaceDimension}

Constraints

None.

9.2.5.2.5 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.

9.2.5.2.6 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.

9.2.5.2.7 Object

Element

Structure

Description

An *`Object`* is an *`Occurrence`* that is not a *`Performance`*. It is the most general *`Structure`*. All other *`Structures`* specialize it directly or indirectly.

General Types

Occurrence

Features

enactedPerformances : Performance [0..*] {subsets timeEnclosedOccurrences, involvingPerformances}

Performances that are enacted by this object.

involvingPerformances : Performance [0..*]

Performances in which this Object is involved.

ownedPerformances : Performance [0..*] {subsets timeEnclosedOccurrences, involvingPerformances, suboccurrences}

Performances that are owned by this *Object*. The owning *Object* is the default *this* reference for all *ownedPerformances*.

structuredSpaceBoundary : StructuredSpaceObject [0..1] {subsets spaceBoundary}

A `spaceBoundary` that is a StructuredSpaceObject.

subobjects : Object [0..*] {subsets suboccurrences}

The *suboccurrences* of this *Object* that are also *Objects*.

Constraints

None.

9.2.5.2.8 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.

9.2.5.2.9 Point

Element

Structure

Description

Objects of `innerSpaceDimension` 0.

General Types

Object

Features

innerSpaceDimension : Integer {redefines innerSpaceDimension}

Constraints

None.

9.2.5.2.10 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 meet when this StructuredSpaceObject `isClosed` (`faces` meet at their `edges` and/or `vertices`, while `edges` meet 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.

9.2.5.2.11 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.

9.2.6 Performances

9.2.6.1 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 8.3.4.6 and compare to *Objects* in 9.2.5). *Performances* can coordinate others that *HappenDuring* them, identified as their *`subperformances`* (see Steps in 8.3.4.6 and 8.4.4.7). *Performances* also coordinate and potentially affect other things, some of which might come into existence (start, be "created") or cease to exist (end, be "destroyed") during a Performance, and some that might be used without being affected at all ("catalysts"). Some of these other things 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 (as `parameters`, see 8.3.4.6).

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 8.3.4.7). 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 8.3.4.8 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 8.3.4.8).

9.2.6.2 Elements

9.2.6.2.1 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.

9.2.6.2.2 booleanEvaluations

Element

BooleanExpression

Description

`booleanEvaluations` is a specialization of `evaluations` restricted to type BooleanEvaluation.

General Types

BooleanEvaluation

evaluations

Features

None.

Constraints

None.

9.2.6.2.3 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.

9.2.6.2.4 evaluations

Element

Expression

Description

`evaluations` is a specialization of `performances` for Evaluations of functions.

General Types

performances
Evaluation

Features

None.

Constraints

None.

9.2.6.2.5 falseEvaluations

Element

BooleanExpression

Description

`falseEvaluations` is a subset of `booleanEvaluations` that result in false. It is the most general Feature of Invariants that are negated.

General Types

booleanEvaluations

Features

[no name] : LiteralEvaluation

Constraints

None.

9.2.6.2.6 Involves

Element

Association

Description

Involves classifies relationships between Performances and Objects.

General Types

None.

Features

None.

Constraints

None.

9.2.6.2.7 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.

9.2.6.2.8 literalEvaluations

Element

Expression

Description

`literalEvaluations` is a specialization of `evaluations` restricted to type LiteralEvaluation.

General Types

LiteralEvaluation
evaluations

Features

None.

Constraints

None.

9.2.6.2.9 MetadataAccessEvaluation

Element

Function

Description

MetadataAccessEvaluation is a specialization of Evaluation for the case of MetadataAccessExpressions.

General Types

Evaluation

Features

result : Metaobject [0..*] {redefines result}

The result of this NullEvaluation, which always must be empty (i.e., "null").

Constraints

None.

9.2.6.2.10 metadataAccessEvaluations

Element

Expression

Description

`metadataAccessEvaluations` is a specialization of `evaluations` restricted to type MetadataAccessEvaluation.

General Types

MetadataAccessEvaluation
evaluations

Features

None.

Constraints

None.

9.2.6.2.11 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.

9.2.6.2.12 nullEvaluations

Element

Expression

Description

`nullEvaluations` is a specialization of `evaluations` restricted to type NullEvaluation.

General Types

NullEvaluation
evaluations

Features

None.

Constraints

None.

9.2.6.2.13 Performance

Element

Behavior

Description

A *`Performance`* is an *`Occurrence`* that is not a *`Object`*. It is the most general *`Behavior`*. All other *`Behaviors`* specialize it directly or indirectly.

General Types

Occurrence

Features

enclosedPerformances : Performance [0..*] {subsets timeEnclosedOccurrences}

`timeEnclosedOccurrences` of this Performance that are also Performances.

involvedObjects : Object [0..*]

Objects that are involved in this Performance.

performers : Object [0..*] {subsets involvedObjects}

Objects that enact this performance.

subperformances : Performance [0..*] {subsets enclosedPerformances, suboccurrences} enclosedPerformances that are composite. The default this context of a subperformance is by default the same as that of its owning Performance. This means that the context for any Performance that is in a composition tree rooted in a Performance that is not itself owned by an Object is the root Performance. If the root Performance is an ownedPerformance of an Object, then that Object is the context.

thisPeformance : Performance

The "context" Performance during which this Performance takes place. It defaults to the root of the subperformances composition tree. It is the default dispatchScope for Performances.

Constraints

None.

9.2.6.2.14 performances

Element

Step

Description

performances is the most general feature for Performances of behaviors.

General Types

Performance
things

Features

None.

Constraints

None.

9.2.6.2.15 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.

9.2.6.2.16 trueEvaluations

Element

BooleanExpression

Description

`trueEvaluations` is a subset of `booleanEvaluations` that result in true. It is the most general Feature of Invariants that are not negated.

General Types

booleanEvaluations

Features

[no name] : LiteralEvaluation

Constraints

None.

9.2.7 Transfers

9.2.7.1 Transfers Overview

Transfers are *Performances* and *BinaryLinks* that carry *`items`* from their *`source`* *Occurrence* to their *`target`* *Occurrence*. *FlowTransfers* are *Transfers* that start by "picking up" their *`items`* from the *`sourceOutput`* Feature (or one of its redefinitions) of the *`source`* and end with "dropping them off" at the *`targetInput`* Feature of the *`target`* (or one of its redefinitions, see 8.3.3.1.5 about outputs and inputs). *FlowTransfers* do this by specifying the existence of *BinaryLinkObjects* between their *`source`* / *`target`* and values of *`sourceOutput`* / *`targetInput`* Features of those, identified by the Connectors *`sourceOutputLink`* and *`targetOutputLink`*, respectively (these can be redefined to specialized associations when *FlowTransfer* is used). Each *`sourceOutputLink`* identifies an output as its *`transferPayload`* (one of the values of *`sourceOutput`* on the *`source`* at the time a *FlowTransfer* starts) . Each *`targetInputLink`* identifies an input also as its *`transferPayload`* (one of the values of *`targetInput`* on the *`target`* at the time a *Transfer* ends). Both collections of *`transferPayloads`* are the same as the *FlowTransfer*'s *`items`*, and do not change while it is carried out.

Transfers are required to take zero time when their *isInstant* Feature is *`true`* (*`startShot`* and *`endShot`* are the same, see Portions and Time Slices in 9.2.4.1), otherwise they might take time to carry out.

Two Boolean Features of *FlowTransfers* affect timing of their *`sourceOutputLinks`* and *`targetOutputLinks`*:

- *`isMove true`* requires *`sourceOutputLinks`* to end (cease to exist) when the *Transfer* starts, otherwise the *Transfer* has no effect on the *`sourceOutputLinks`*.
- *`isPush true`* requires the *Transfer* to start when its *`sourceOutputLinks`* do (begin to exist), otherwise the *Transfer* can start anytime after the *`sourceOutputLinks`* do.

MessageTransfers are *Transfers* that do not have the additional capabilities of FlowTransfers. *SendPerformances* and *AcceptPerformances* are *Performances* for specifying when *MessageTransfers* come into and go out of *Occurrences*, respectively. *SendPerformances* require a *MessageTransfer* as *`outgoingTransferFromSelf`* from a designated `sender` (defaulting to *`this`*, see 9.2.4.2.13), carrying a *`sentItem`*, optionally to a designated *`receiver`*. *AcceptPerformances* require an *`incomingTransferToSelf`* to a designated *`receiver`* (defaulting to *`this`*), carrying an *`acceptedItem`*.

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 8.3.4.9). *Transfer* is not the most general binary Interaction, and *`transfers`* is not the most general feature typed by binary Interactions, because binary Interactions can include more than one ItemFlow, as well as other Interactions.

ItemFlow's `itemType` gives the kind of things being transferred (most generally the type of *`item`*, above). For *FlowTransfers*, ItemFlow's `sourceOutputFeature` and `targetInputFeature` specify which Features of its connected Feature *Occurrences* identify outputs and inputs, respectively (most generally *`sourceOutput`* and *`targetInput`* above, respectively).

9.2.7.2 Elements

9.2.7.2.1 AcceptPerformance

Element

Behavior

Description

AcceptPerformances are Performances that require an `incomingTransferToSelf` of a designated `receiver` Occurrence (defaulting to `this`), providing an `acceptedItem` as output.

General Types

Performance

Features

acceptedItem : Anything [0..*]

acceptedTransfer : MessageTransfer [0..1] {subsets receiver.incomingTransfersToSelf}

receiver : Occurrence

receiver.incomingTransfersToSelf : Transfer [0..*]

Constraints

None.

9.2.7.2.2 FlowTransfer

Element

Interaction

Description

A FlowTransfer is a Transfer identifying an output feature of the `source` to pick up items from and an input feature of the `target` to drop them off. They can start when items are available at the `source` and move or copy them to the `target`.

General Types

Transfer

Features

[no name] : Occurrence

[no name] : Occurrence

isMove : Boolean

isPush : Boolean

sourceOutputLink : BinaryLinkObject [1..*]

targetInputLink : BinaryLinkObject [1..*]

Constraints

None.

9.2.7.2.3 FlowTransferBefore

Element

Interaction

Description

General Types

TransferBefore
FlowTransfer

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.7.2.4 flowTransfers

Element

Feature

Description

General Types

transfers
FlowTransfer

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.7.2.5 flowTransfersBefore

Element

Feature

Description

General Types

transfersBefore
FlowTransferBefore

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.7.2.6 MessageTransfer

Element

Interaction

Description

A MessageTransfer is a Transfer that does not specify where items are picked up and dropped off (see FlowTransfer). They are sent by SendPerformances and accepted by AcceptPerformances.

General Types

Transfer

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.7.2.7 messageTransfers

Element

Feature

Description

General Types

transfers
MessageTransfer

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.7.2.8 SendPerformance

Element

Behavior

Description

SendPerformances are Performances that require an `outgoingTransferFromSelf` from a designated `sender` Occurrence (defaulting to `this`), carrying a given `sentItem`, optionally to a designated `receiver`.

General Types

Performance

Features

receiver : Occurrence [0..1]

receiver.incomingTransfersToSelf : Transfer [0..*]

sender : Occurrence

sender.outgoingTransfersToSelf : Transfer [0..*]

sentItem : Anything [0..*]

sentTransfer : MessageTransfer {subsets sender.outgoingTransfersToSelf}

Constraints

None.

9.2.7.2.9 Transfer

Element

Interaction

Description

A Transfer is a Performance and BinaryLink that carries `items` from its `source` to its `target`.

General Types

Performance
BinaryLink

Features

isInstant : Boolean

item : Anything [1..*]

self : Transfer {redefines self}

source : Occurrence {redefines source}

sourceSendShot : Occurrence

target : Occurrence {redefines target}

targetReceiveShot : Occurrence

toTransferSources : Occurrence [0..*] {subsets toSources}

toTransferTargets : Occurrence [0..*] {subsets toTargets}

Constraints

None.

9.2.7.2.10 TransferBefore

Element

Interaction

Description

A TransferBefore is Transfer that happens after its `source` and before its `target`.

General Types

Transfer
HappensBefore

Features

source : Occurrence {redefines earlierOccurrence, source}

target : Occurrence {redefines laterOccurrence, target}

toTransferSources : Occurrence [0..*] {redefines predecessors, toTransferSources}

toTransferTargets : Occurrence [0..*] {redefines toTransferTargets, successors}

Occurrences whose input is the target of a TransferBefore of items from this Occurrence.

Constraints

None.

9.2.7.2.11 transfers

Element

Feature

Description

General Types

Transfer

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.7.2.12 transfersBefore

Element

Feature

Description

General Types

TransferBefore
transfers

Features

[no name] : Occurrence

[no name] : Occurrence

Constraints

None.

9.2.8 Feature Referencing Performances

9.2.8.1 Feature Referencing Performances Overview

The `FeatureReferencingPerformances` 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.

9.2.8.2 Elements

9.2.8.2.1 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.

9.2.8.2.2 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

```
isEmpty(monitor2) == (monitor1.afterValues == isToTrue)
```

9.2.8.2.3 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.

9.2.8.2.4 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.

9.2.8.2.5 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 snapshots}

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

```
not beforeValues == afterValues
```

9.2.8.2.6 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.

9.2.8.2.7 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.

9.2.8.2.8 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.

9.2.9 Control Performances

9.2.9.1 Control Performances Overview

The *ControlPerformances* package defines Behaviors 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.
- be included in a Feature of its `featuringBehavior` that unions (see 7.3.2.7) all the outgoing Successions, and is bound to the *`outgoingHBLink`* of the Step (see 7.3.4.6 on feature chaining).

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` that unions all the incoming Successions, and is bound to the *`incomingHBLink`* of the Step.

IfPerformances are *Performances* that determine whether a clause occurs based on the result of a *BooleanEvaluation* (see 9.2.6.1). Two specializations *IfThenPerformance* and *IfElsePerformance* have one clause each, `thenClause` and `elseClause`, respectively, that occur when the *BooleanEvaluation* is `true` or `false`, respectively. *IfThenElsePerformance* specializes both of these to have two clauses.

LoopPerformances are *Performances* with a `body` that occurs iteratively as determined by *BooleanEvaluations* `whileTest` and `untilTest`. The `body` occurs repeatedly in sequence (iteratively) as long as the result of `whileTest` is `true` before each iteration (and after the previous one, if any), and the result of `untilTest` is `false` after each iteration and before the next one (except after the last one, when it is `false`).

9.2.9.2 Elements

9.2.9.2.1 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

Specializations subset this from the union of all successions going out of a decision step.

Constraints

None.

9.2.9.2.2 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.

9.2.9.2.3 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.

9.2.9.2.4 IfThenElsePerformance

Element

Behavior

Description

An IfThenElsePerformance is an IfThenPerformance and an IfElsePerformance.

General Types

IfElsePerformance
IfThenPerformance

Features

None.

Constraints

None.

9.2.9.2.5 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.

9.2.9.2.6 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.

9.2.9.2.7 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

Specializations subset this from the union of all successions coming into a merge step.

Constraints

None.

9.2.10 Transition Performances

9.2.10.1 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 (see 7.4.6.4) going out of an *Occurrence* Feature (`Succession::sourceFeature`) has values (*HappensBefore* links), based on values of `sourceFeature` (*Occurrences*) 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`, 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:

- identify at least one *Transfer* as `trigger` that `targets` `triggerTarget`.
- 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 9.2.11.1).
- 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. *NonStateTransitionPerformances* require their `guards` to happen after `transitionLinkSource` (see 9.2.11.1 about StateTransitionPerformances).

9.2.10.2 Elements

9.2.10.2.1 NonStateTransitionPerformance

Element

Behavior

Description

General Types

TransitionPerformance

Features

isTriggerAfter : Boolean

Constraints

None.

9.2.10.2.2 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.

9.2.10.2.3 TransitionPerformance

Element

Behavior

Description

General Types

Performance

Features

accept : AcceptPerformance [0..1] {subsets enclosedPerformances}

effect : Performance [0..*] {subsets enclosedPerformances}

guard : Evaluation [0..*] {subsets enclosedPerformances}

guardConstraint : TPCGuardConstraint [0..*]

transitionLink : HappensBefore [0..1]

transitionLinkSource : Performance

trigger : MessageTransfer [0..*]

triggerTarget : Occurrence

Constraints

None.

9.2.11 State Performances

9.2.11.1 State Performances Overview

The *StatePerformance* package contains a library model for the semantics of state-based behavior, including *StatePerformances* and *StateTransitionPerformances*.

StatePerformances are *DecisionPerformances* (see 9.2.9.1) that

- only have Steps defined in this library, or specialized from them.
- can identify *Transfers* that might be followed by taking the last of the above Steps (see `exit` below).

Usages of *StatePerformance* can specialize its library-defined Steps to specify how they are carried out, as well as how the *Transfers* above are identified. Any Behavior can use (have `steps` typed by) *StatePerformances*, not only "state machines".

The *StatePerformance* Steps defined in this library are:

- `entry` [1]: happens before all *Performances* of `middle`.
- `middle` [1..*]: happen before the `exit` *Performance* (see below). Additional modeler-defined Steps must subset this one.
- `do` [1]: a `middle` *Performance* that starts before the others.
- `exit` [1]: happens after the end of *Transfers* identified by the *StatePerformance* (see `acceptable` below).

StatePerformances identify *Transfers* that happen before (potentially "trigger") their `exit` with these Features:

- `acceptable` [*]: candidates for being identified as `accepted`.
- `accepted` [0..1]: one of the `acceptable` transfers that enables `exit` to start. This must have a value if `acceptable` does.

The `accepted` *Transfer* must end (arrive) during a *StatePerformance* when its `isTriggerDuring` is `true`.

StateTransitionPerformances are one way to determine which *Transfers* are `acceptable` to a *StatePerformance*. They are *TransitionPerformances* (see 9.2.10.1) that

- have a *StatePerformance* as their `transitionLinkSource`.
- are the type of Steps connected to Successions (see 7.4.6.4) going out of a *StatePerformance* Step (as in "state machines").

StateTransitionPerformances identify *MessageTransfers* (see 9.2.7.1) by these Features:

- `acceptable` [*]: candidates for being identified as `trigger`. This subsets `acceptable` of their `transitionLinkSource`.
- `trigger` [0..1]: one of the `acceptable` transfers. This subsets `accepted` of their `transitionLinkSource`.

The *trigger* *Transfer* must end (arrive) during the *transitionLinkSource* when *StateTransitionPerformance* *isTriggerDuring* is *true*.

The Subsettings above enable a *StatePerformance* Step to constrain all the *StateTransitionPerformance*s Steps connected to its outgoing Successions, including to decide which of the *MessageTransfers* *acceptable* to those *StateTransitionPerformances* will be *accepted* by the *StatePerformance* and *trigger* which outgoing Succession (will have a *HappenBeforeLink* value).

StateTransitionPeformances require their *guards* to happen after the *nonDoMiddle* Step of the *transitionLinkSource* (all the *middle* *Performances* except for *do*) and before the *exit* Step (compare to *NonStateTransitionPerformances* in 9.2.10.1).

StatePerformances identify the *Transfer* that triggered a transition into it (a *StateTransitionPeformance* *trigger*), if any, by the Feature *incomingTransitionTrigger*.

Some Features of *Occurrences* constrain *StatePerformances* and *TransitionPerformances* composed under them, as sometimes needed in state machines:

- *incomingTransferSort* determines which *Transfer* should be *accepted* when multiple are *acceptable* ones, by comparing two *Transfers* at a time. It defaults to earlierFirstIncomingTransferSort for *Occurrences*, including *StatePerformances*, which is *true* if the first *Transfer* ends (arrives) before the other.
- *isDispatch* being *true* prevents the same *Transfer* from being *accepted* more than once by *StatePeformances* composed under *dispatchScope*, and prevents from being *accepted* at all any *acceptable* *Transfers* that are not *accepted* and are higher in *incomingTransferSort* order than the one that is. It defaults to *true* for *Performances*, including *StatePerformances*, and *false* for other *Occurrences*, while *dispatchScope* defaults to *thisPerformance* for *StatePerformances*, the top *Performance* (indirectly) composing the *StatePerformance* (see 9.2.6.2.13), and *self* for other *Occurrences* (see 9.2.2.1).
- *isRunToCompletion* being *true* prevents *TransitionPerformances* composed under *runToCompletionScope* from happening during *entry*. It defaults to the same as it is on *this* for *StatePerformances*, the *Object* directly composing *thisPerformance*, or *thisPerformance* if there is none (see 9.2.4.2.13), and *true* for other *Occurrences*, while *runToCompletionScope* defaults to the same as it is on *this* for *StatePerformances*, and *self* for other *Occurrences*.

9.2.11.2 Elements

9.2.11.2.1 StatePerformance

Element

Behavior

Description

General Types

DecisionPerformance

Features

acceptable : MessageTransfer [0..*] {union}

accepted : MessageTransfer [0..1] {subsets acceptable} deferrable : MessageTransfer [0..*] {subsets acceptable}

do : Performance {subsets middle}

entry : Performance {subsets timeEnclosedOccurrences}

exit : Performance {subsets timeEnclosedOccurrences}

incomingTransitionTrigger : MessageTransfer [0..1]

Transfer that triggered a transition into this state performance.

isTriggerDuring : Boolean

/middle : Performance [1..*] {subsets timeEnclosedOccurrences, union}

/nonDoMiddle : Performance [0..*] {subsets middle}

Constraints

None.

9.2.11.2.2 StateTransitionPerformance

Element

Behavior

Description

General Types

TransitionPerformance

Features

acceptable : MessageTransfer [0..*] {subsets triggerTarget.incomingTransfersToSelf, transitionLinkSource.acceptable}

isTriggerDuring : Boolean

transitionLinkSource : StatePerformance {redefines transitionLinkSource}

transitionLinkSource.acceptable : MessageTransfer [0..*]

transitionLinkSource.accepted : MessageTransfer [0..1]

transitionLinkTarget : Occurrence [0..1]

trigger : MessageTransfer [0..1] {subsets acceptable, transitionLinkSource.accepted, redefines trigger}

triggerTarget.incomingTransfersToSelf : Transfer [0..*]

Constraints

None.

9.2.12 Clocks

9.2.12.1 Clocks Overview

This package models *Clocks* that provide an advancing numerical reference usable for quantifying the time of an *Occurrence*.

9.2.12.2 Elements

9.2.12.2.1 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.

9.2.12.2.2 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}

Default is inherited *Occurrence::localClock*.

duration : Real {redefines duration}

o : Occurrence {redefines o}

Constraints

None.

9.2.12.2.3 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}

Default is inherited `Occurrence::localClock`.

o : Occurrence {redefines o}

timeValue : Real {redefines timeInstant}

Constraints

None.

9.2.12.2.4 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 : NumericalValue

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*.

9.2.12.2.5 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

Default is inherited *Occurrence::localClock*.

duration : NumericalValue

o : Occurrence

Constraints

None.

9.2.12.2.6 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

Default is inherited *Occurrence::localClock*.

o : Occurrence

timeInstant : NumericalValue

Constraints

startTimeConstraint

The *TimeOf* an *Occurrence* is equal to the time of its start snapshot.

timeContinuityConstraint

If one *Occurrence* happens immediately before another, then the *TimeOf* the end snapshot of the first Occurrence equals the *TimeOf* the second *Occurrence*.

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*.

9.2.12.2.7 universalClock

Element

Feature

Description

universalClock is a single *Clock* that can be used as a default universal time reference.

General Types

Clock
objects

Features

None.

Constraints

None.

9.2.13 Observation

9.2.13.1 Observation Overview

This package models a framework for monitoring *Boolean* conditions and notifying registered observers when they change from false to true.

9.2.13.2 Elements

9.2.13.2.1 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.

9.2.13.2.2 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.

9.2.13.2.3 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

signalCondition : BooleanEvaluation

A BooleanExpression whose result is being monitored.

signalMonitor : ChangeMonitor

The *ChangeMonitor* responsible for monitoring the *signalCondition*.

Constraints

None.

9.2.13.2.4 defaultMonitor

Element

Feature

Description

defaultMonitor is a single *ChangeMonitor* that can be used as a default.

General Types

ChangeMonitor
objects

Features

None.

Constraints

None.

9.2.13.2.5 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.

9.2.13.2.6 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.

9.2.14 Triggers

9.2.14.1 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.

9.2.14.2 Elements

9.2.14.2.1 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.

9.2.14.2.2 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 *localClock*, which will be bound when the function is invoked.

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.

9.2.14.2.3 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 *timeInstant*. The default is the *localClock*, which will be bound when the function is invoked.

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

timeInstant : NumericalValue

The time instant, relative to the *clock*, at which the *TimeSignal* should be sent.

Constraints

None.

9.2.14.2.4 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.

9.2.15 SpatialFrames

9.2.15.1 SpatialFrames Overview

This package models spatial frames of reference for quantifying the position of points in a three-dimensional space.

9.2.15.2 Elements

9.2.15.2.1 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.

9.2.15.2.2 CartesianCurrentPositionOf

Element

Function

Description

The `CurrentPositionOf` a Point relative to a `CartesianSpatialFrame` is a `CartesianThreeVectorValue`.

General Types

CurrentPositionOf

Features

clock : Clock {redefines clock}

Defaults to the `localClock of the frame`.

frame : CartesianSpatialFrame {redefines frame} point : Point {redefines point}

positionVector : CartesianThreeVectorValue {redefines positionVector}

Constraints

None.

9.2.15.2.3 CartesianDisplacementOf

Element

Function

Description

The `DisplacementOf` two Points relative to a `CartesianSpatialFrame` is a `CartesianThreeVectorValue`.

General Types

DisplacementOf

Features

clock : Clock {redefines clock}

Defaults to the `localClock of the frame`.

displacementVector : CartesianThreeVectorValue {redefines displacementVector}

frame : CartesianSpatialFrame {redefines frame}

point1 : Point {redefines point1}

point2 : Point {redefines point2}

time : NumericalValue {redefines time}

Constraints

None.

9.2.15.2.4 CartesianPositionOf

Element

Function

Description

The `PositionOf` a Point relative to a `CartesianSpatialFrame` is a `CartesianThreeVectorValue`.

General Types

PositionOf

Features

clock : Clock {redefines clock}

Defaults to the *localClock of the frame*.

frame : CartesianSpatialFrame {redefines frame}

point : Point {redefines point}

positionVector : CartesianThreeVectorValue {redefines positionVector}

time : NumericalValue {redefines time}

Constraints

None.

9.2.15.2.5 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.

9.2.15.2.6 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

Defaults to the *localClock of the frame*.

displacementVector : ThreeVectorValue

frame : SpatialFrame

point1 : Point

point2 : Point

Constraints

None.

9.2.15.2.7 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

Defaults to the *localClock of the frame*.

frame : SpatialFrame

point : Point

positionVector : ThreeVectorValue

Constraints

None.

9.2.15.2.8 defaultFrame

Element

Feature

Description

defaultFrame is a fixed *SpatialFrame* used as a universal default.

General Types

SpatialFrame

Features

None.

Constraints

None.

9.2.15.2.9 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

Defaults to the *localClock* of the *frame*.

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)
```

9.2.15.2.10 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

Defaults to the *localClock* of the *frame*.

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)
}
```

9.2.15.2.11 SpatialFrame

Element

Structure

Description

General Types

Body

Features

None.

Constraints

None.

9.2.16 Metaobjects

9.2.16.1 Metaobjects Overview

This package defines Metaclasses and Features that are related to the typing of syntactic and semantic metadata.

9.2.16.2 Elements

9.2.16.2.1 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.

9.2.16.2.2 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.

9.2.16.2.3 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.

9.2.17 KerML

This package contains a reflective KerML model of the KerML abstract syntax. It is generated from the normative MOF abstract syntax model (see 8.3) as follows.

1. The *KerML* model contains subpackages for *Root*, *Core,* and *Kernel,* but all elements are also imported into the top-level package, so they can be referenced directly from the *KerML* namespace.

2. A metaclass from the MOF model is mapped into a `Metaclass` in the KerML package.
    - The MOF metaclass name is mapped unchanged.
    - Generalizations of the MOF metaclass are mapped to `ownedSpecializations`.
    - All properties from the MOF metaclass are mapped to `features` of the corresponding KerML `Metaclass` (see below). All non-association-end properties are grouped before association-end properties.
3. A property from the MOF model is mapped into a `Feature`.
    - The following feature properties are set as appropriate:
        - `isAbstract = true` if the MOF property is a derived union
        - `isComposite = true` if the MOF property is composite.
        - `isReadonly = true` if the MOF property is read-only.
        - `isDerived = true` if the MOF property is derived.
    - The MOF property name is mapped unchanged.
    - The MOF property type is mapped to an `ownedTyping` relationship.
        - If the MOF property type is a primitive type, the relationship is to the corresponding type from the *`ScalarValues`* package (see 9.3.2).
        - If the MOF property type is a metaclass, the relationship is to the corresponding reflective `Metaclass`.
    - The MOF property multiplicity is mapped to an owned `MultiplicityRange` with bounds given by `LiteralExpressions`.
    - Subsetted properties from the MOF property are mapped to `ownedSubsettings` of the corresponding reflective `Features`.
    - Redefined properties from the MOF property are mapped to `ownedRedefinitions` of the corresponding reflective `Features`.
    - If the MOF property is `annotatedElement`, then `Metaobject::annotatedElement` is added to the list of redefined properties for the mapping.
4. An enumeration from the MOF model is mapped into a `DataType`.
    - The MOF enumeration name is mapped unchanged.
    - Each enumeration literal from the MOF enumeration is mapped into a `ownedMember Feature` (*not* and `ownedFeature`).
        - The MOF enumeration literal name is mapped unchanged.
        - The member `Feature` is given an owned `MultiplicityRange` of `1..1`.

Note that associations are not mapped from the MOF model and, hence, non-navigable association-owned end properties are not included in the reflective model.

9.3 Data Type Library

9.3.1 Data Types Library Overview

The Data Types Library provides a standard set of commonly used `DataTypes` for scalar, vector and collection values.

9.3.2 Scalar Values

9.3.2.1 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.

9.3.2.2 Elements

9.3.2.2.1 Boolean

Element

DataType

Description

Boolean is a ScalarValue type whose instances are true and false.

General Types

ScalarValue

Features

None.

Constraints

None.

9.3.2.2.2 Complex

Element

DataType

Description

Complex is the type of complex numbers.

General Types

Number

Features

None.

Constraints

None.

9.3.2.2.3 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.

9.3.2.2.4 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.

9.3.2.2.5 Number

Element

DataType

Description

Number is the base type for all NumericalValue types that represent numbers.

General Types

NumericalValue

Features

None.

Constraints

None.

9.3.2.2.6 NumericalValue

Element

DataType

Description

NumericalValue is the base type for all ScalarValue types that represent numerical values.

General Types

ScalarValue

Features

None.

Constraints

None.

9.3.2.2.7 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.

9.3.2.2.8 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.

9.3.2.2.9 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.

9.3.2.2.10 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.

9.3.2.2.11 String

Element

DataType

Description

String is a ScalarValue type whose instances are strings of characters.

General Types

ScalarValue

Features

None.

Constraints

None.

9.3.3 Collections

9.3.3.1 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.

9.3.3.2 Elements

9.3.3.2.1 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` 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

```
flattenedSize == size(elements)
```

9.3.3.2.2 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.

9.3.3.2.3 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.

9.3.3.2.4 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.

9.3.3.2.5 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.

9.3.3.2.6 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.

9.3.3.2.7 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.

9.3.3.2.8 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.

9.3.3.2.9 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.

9.3.3.2.10 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.

9.3.3.2.11 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.

9.3.4 Vector Values

9.3.4.1 Vector Values Overview

9.3.4.2 Elements

9.3.4.2.1 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.

9.3.4.2.2 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.

9.3.4.2.3 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.

9.3.4.2.4 ThreeVectorValue

Element

DataType

Description

A *ThreeVectorValue* is a *NumericalVectorValue* that has dimension 3.

General Types

NumericalVectorValue

Features

dimension : Positive [0..*] {redefines elements}

Constraints

None.

9.3.4.2.5 VectorValue

Element

DataType

Description

A *VectorValue* is an abstract data type whose values may be operated on using *VectorFunctions*.

General Types

None.

Features

None.

Constraints

None.

9.4 Function Library

The Function Library includes library models of basic `Functions` that operate on `DataTypes` from the Data Type Library (see 9.3). The KerML operator expression notation translates 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` can be used with these specialized `Functions` and `DataTypes`, extended with domain-specific semantics.

9.4.1 Function Library Overview

The Function Library includes library models of basic `Functions` that operate on `DataTypes` from the Data Type Library (see 9.3). The KerML operator expression notation translates 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` can be used with these specialized `Functions` and `DataTypes`, extended with domain-specific semantics.

9.4.2 Base Functions

9.4.2.1 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.

9.4.2.2 Elements

```
abstract function '=='{
    in x: Anything[0..1];
    in y: Anything[0..1];
    return : Boolean[1];
}

function '!='{
    in x: Anything[0..1];
    in y: Anything[0..1];
    return : Boolean[1] = not (x == y);
}

abstract function '==='{
    in x: Anything[0..1];
    in y: Anything[0..1];
    return : Boolean[1];
}

function '!=='{
    in x: Anything[0..1];
    in y: Anything[0..1];
    return : Boolean[1] = not (x === y);
}

function ToString{
    in x: Anything[0..1];
    return : String;
}

function '['{
    in x: Anything[0..*] nonunique;
    in y: Anything[0..*] nonunique;
    return : Anything[0..*] nonunique;
}

function '#'{
    in seq: Anything[0..*] ordered nonunique;
    in index: Positive[1..*] ordered nonunique;
    return : Anything[0..1];
}
```

```
function ','{
    in seq1: Anything[0..*] ordered nonunique;
    seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}

abstract function 'all'{
     return : Object[0..*];
}

abstract function 'istype'{
    in seq: Anything[0..*];
    abstract feature 'type': Anything;
    return : Boolean[1];
}

abstract function 'hastype'{
    in seq: Anything[0..*];
    abstract feature 'type': Anything;
    return : Boolean;
}

abstract function '@'{
    in seq: Anything[0..*];
    abstract feature 'type': Anything;
    return : Boolean[1];
}

abstract function '@@'{
    in seq: Metaobject[0..*];
    abstract feature 'type': Metaobject;
    return : Boolean[1];
}

abstract function 'as'{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}

abstract function 'meta'{
    in seq: Metaobject[0..*] ordered nonunique;
    return : Metaobject[0..*] ordered nonunique;
}
```

9.4.3 Data Functions

9.4.3.1 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.

9.4.3.2 Elements

```
abstract function '==' specializes BaseFunctions::'=='
    { in x: DataValue[0..1]; in y: DataValue[0..1]; return : Boolean[1]; }
abstract function '===' specializes BaseFunctions::'==='
    { in x: DataValue[0..1]; in y: DataValue[0..1]; return : Boolean[1]; }

abstract function '+'
```

```
    { in x: DataValue[1]; in y: DataValue[0..1]; return : DataValue[1]; }
abstract function '-'
    { in x: DataValue[1]; in y: DataValue[0..1]; return : DataValue[1]; }
abstract function '*'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '/'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '**'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '^'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '%'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function 'not'
    { in x: DataValue[1]; return : DataValue[1]; }
abstract function 'xor'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function '~'
    { in x: DataValue[1]; return : DataValue[1]; }
abstract function '|'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '&'
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function '<'
    { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }
abstract function '>'
    { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }
abstract function '<='
    { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }
abstract function '>='
    { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }

abstract function Max
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function Min
    { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function '..'
    { in lower: DataValue[1]; in upper: DataValue[1]; return : DataValue[0..*] ordered; }
```

9.4.4 Scalar Functions

9.4.4.1 Scalar Functions Overview

This package defines abstract Functions that specialize the DataFunctions for use with ScalarValues.

9.4.4.2 Elements

```
abstract function '+' specializes DataFunctions::'+'
    { in x: ScalarValue[1]; in y: ScalarValue[0..1]; return : ScalarValue[1]; }
abstract function '-' specializes DataFunctions::'-'
    { in x: ScalarValue[1]; in y: ScalarValue[0..1]; return : ScalarValue[1]; }
abstract function '*' specializes DataFunctions::'*'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '/' specializes DataFunctions::'/'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
```

```
abstract function '**' specializes DataFunctions::'**'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '^' specializes DataFunctions::'^'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '%' specializes DataFunctions::'%'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

abstract function 'not' specializes DataFunctions::'not'
    { in x: ScalarValue[1]; return : ScalarValue[1]; }
abstract function 'xor' specializes DataFunctions::'xor'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

abstract function '~' specializes DataFunctions::'~'
    { in x: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '|' specializes DataFunctions::'|'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '&' specializes DataFunctions::'&'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

abstract function '<' specializes DataFunctions::'<'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }
abstract function '>' specializes DataFunctions::'>'
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }
abstract function '<=' specializes DataFunctions::'<='
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }
abstract function '>=' specializes DataFunctions::'>='
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }

abstract function max specializes DataFunctions::Max
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function min specializes DataFunctions::Min
    { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

abstract function '..' specializes DataFunctions::'..'
    { in lower: ScalarValue[1]; in upper: ScalarValue[1];
      return : ScalarValue[0..*]; }
```

9.4.5 Boolean Functions

9.4.5.1 Boolean Functions Overview

This package defines Functions on Boolean values, including those corresponding to (non-conditional) logical operators in the KerML expression notation.

9.4.5.2 Elements

```
function 'not' specializes ScalarFunctions::'not'
    { in x: Boolean[1]; return : Boolean[1]; }
function 'xor' specializes ScalarFunctions::'xor'
    { in x: Boolean[1]; in y: Boolean[1]; return : Boolean[1]; }

function '|' specializes ScalarFunctions::'|'
    { in x: Boolean[1]; in y: Boolean[1]; return : Boolean[1]; }
function '&' specializes ScalarFunctions::'&'
    { in x: Boolean[1]; in y: Boolean[1]; return : Boolean[1]; }

function '==' specializes DataFunctions::'=='
    { in x: Boolean[0..1]; in y: Boolean[0..1]; return : Boolean[1]; }
```

```
function ToString specializes BaseFunctions::ToString
    { in x: Boolean[1]; return : String[1]; }
function ToBoolean
    { in x: String[1]; return : Boolean[1]; }
```

9.4.6 String Functions

9.4.6.1 String Functions Overview

This package defines Functions on String values, including those corresponding to string concatenation and comparison operators in the KerML expression notation.

9.4.6.2 Elements

```
function '+' specializes ScalarFunctions::'+'
    { in x: String[1]; in y:String[1]; return : String[1]; }

function Length
    { in x: String[1]; return : Natural[1]; }
function Substring
    { in x: String[1]; in lower: Integer[1]; in upper: Integer[1];
      return : String[1]; }

function '<' specializes ScalarFunctions::'<'
    { in x: String[1]; in y: String[1]; return : Boolean[1]; }
function '>' specializes ScalarFunctions::'>'
    { in x: String[1]; in y: String[1]; return : Boolean[1]; }
function '<=' specializes ScalarFunctions::'<='
    { in x: String[1]; in y: String[1]; return : Boolean[1]; }
function '>=' specializes ScalarFunctions::'>='
    { in x: String[1]; in y: String[1]; return : Boolean[1]; }

function '==' specializes DataFunctions::'=='
    { in x: String[0..1]; in y: String[0..1]; return : Boolean[1]; }

function ToString specializes BaseFunctions::ToString
    { in x: String[1]; }
```

9.4.7 Numerical Functions

9.4.7.1 Numerical Functions Overview

This package defines abstract Functions on Numerical values for general arithmetic and comparison operations.

9.4.7.2 Elements

```
abstract function isZero
    { in x: NumericalValue[1]; return : Boolean; }
abstract function isUnit
    { in x : NumericalValue[1]; return : Boolean; }

abstract function abs
    { in x: NumericalValue[1]; return : NumericalValue[1]; }

abstract function '+' specializes ScalarFunctions::'+'
    { in x: NumericalValue[1]; in y: NumericalValue[0..1];
      return : NumericalValue[1]; }
abstract function '-' specializes ScalarFunctions::'-'
    { in x: NumericalValue[1]; in y: NumericalValue[0..1];
```

```
      return : NumericalValue[1]; }
abstract function '*' specializes ScalarFunctions::'*'
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }
abstract function '/' specializes ScalarFunctions::'/'
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }
abstract function '**' specializes ScalarFunctions::'**'
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }
abstract function '^' specializes ScalarFunctions::'^'
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }
abstract function '%' specializes ScalarFunctions::'%'
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }

abstract function '<' specializes ScalarFunctions::'<'
    { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }
abstract function '>' specializes ScalarFunctions::'>'
    { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }
abstract function '<=' specializes ScalarFunctions::'<='
    { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }
abstract function '>=' specializes ScalarFunctions::'>='
    { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }

abstract function max specializes ScalarFunctions::max
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }
abstract function min specializes ScalarFunctions::min
    { in x: NumericalValue[1]; in y: NumericalValue[1];
      return : NumericalValue[1]; }

abstract function sum
    { in collection: ScalarValue[0..*]; return : ScalarValue[1]; }
abstract function product
    { in collection: ScalarValue[0..*]; return : ScalarValue[1]; }
```

9.4.8 Complex Functions

9.4.8.1 Complex Functions Overview

This package defines Functions on Complex values, including concrete specializations of the general arithmetic and comparison operations.

9.4.8.2 Elements

```
feature i: Complex[1] = rect(0.0, 1.0);

function rect
    { in re: Real[1]; in im: Real[1]; return : Complex[1]; }
function polar
    { in abs: Real[1]; in arg: Real[1]; return : Complex[1]; }

function re
    { in x: Complex[1]; return : Real[1]; }
function im
    { in x: Complex[1]; return : Real[1]; }

function isZero specializes NumericalFunctions::isZero
```

```
    { in x : Complex[1]; return : Boolean[1]; }
function isUnit specializes NumericalFunctions::isUnit
    { in x : Complex[1]; return : Boolean[1]; }

function abs specializes NumericalFunctions::abs
    { in x: Complex[1]; return : Real[1]; }
function arg
    { in x: Complex[1]; return : Real[1]; }

function '+' specializes NumericalFunctions::'+'
    { in x: Complex[1]; in y: Complex[0..1]; return : Complex[1]; }
function '-' specializes NumericalFunctions::'-'
    { in x: Complex[1]; in y: Complex[0..1]; return : Complex[1]; }
function '*' specializes NumericalFunctions::'*'
    { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }
function '/' specializes NumericalFunctions::'/'
    { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }
function '**' specializes NumericalFunctions::'**'
    { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }
function '^' specializes NumericalFunctions::'^'
    { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }

function '==' specializes DataFunctions::'=='
    { in x: Complex[0..1]; in y: Complex[0..1]; return : Boolean[1]; }

function ToString specializes BaseFunctions::ToString
    { in x: Complex[1]; return : String[1]; }
function ToComplex
    { in x: String[1]; return : Complex[1]; }

function sum specializes NumericalFunctions::sum
    { in collection: Complex[0..*]; return : Complex[1]; }

function product specializes NumericalFunctions::product
    { in collection: Complex[0..*]; return : Complex[1]; }
```

9.4.9 Real Functions

9.4.9.1 Real Functions Overview

This package defines Functions on Real values, including concrete specializations of the general arithmetic and comparison operations.

9.4.9.2 Elements

```
function re :> ComplexFunctions::re
    { in x: Real[1]; return : Real[1] = x; }
function im :> ComplexFunctions::im
    { in x: Real[1]; return : Real[1] = 0.0; }

function abs specializes ComplexFunctions::abs
    { in x: Real[1]; return : Real[1]; }
function arg specializes ComplexFunctions::arg
    { in x: Real[1]; return : Real[1] = 0.0; }

function '+' specializes ComplexFunctions::'+'
    { in x: Real[1]; in y: Real[0..1]; return : Real[1]; }
function '-' specializes ComplexFunctions::'-'
    { in x: Real[1]; in y: Real[0..1]; return : Real[1]; }
function '*' specializes ComplexFunctions::'*'
```

```
    { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function '/' specializes ComplexFunctions::'/'
    { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function '**' specializes ComplexFunctions::'**'
    { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function '^' specializes ComplexFunctions::'^'
    { in x: Real[1]; in y: Real[1]; return : Real[1]; }

function '<' specializes NumericalFunctions::'<'
    { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }
function '>' specializes NumericalFunctions::'>'
    { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }
function '<=' specializes NumericalFunctions::'<='
    { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }
function '>=' specializes NumericalFunctions::'>='
    { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }

function max specializes NumericalFunctions::max
    { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function min specializes NumericalFunctions::min
    { in x: Real[1]; in y: Real[1]; return : Real[1]; }

function '==' specializes ComplexFunctions::'=='
    { in x: Real[0..1]; in y: Real[0..1]; return : Boolean[1]; }

function sqrt
    { in x: Real[1]; return : Real[1]; }

function floor
    { in x: Real[1]; return : Integer[1]; }
function round
    { in x: Real[1]; return : Integer[1]; }

function ToString specializes ComplexFunctions::ToString
    { in x: Real[1]; return : String[1]; }
function ToInteger
    { in x: Real[1]; return : Integer[1]; }
function ToRational
    { in x: Real[1]; return : Rational[1]; }
function ToReal
    { in x: String[1]; return : Real[1]; }

function sum specializes ComplexFunctions::sum
    { in collection: Real[0..*]; return : Real; }

function product specializes ComplexFunctions::product
    { in collection: Real[0..*]; return : Real; }
```

9.4.10 Rational Functions

9.4.10.1 Rational Functions Overview

This package defines Functions on Rational values, including concrete specializations of the general arithmetic and comparison operations.

9.4.10.2 Elements

```
function rat
    { in numer: Integer[1]; in denum: Integer[1]; return : Rational[1]; }
function numer
```

```
    { in rat: Rational[1]; return : Integer[1]; }
function denom
    { in rat: Rational[1]; return : Integer[1]; }

function abs specializes RealFunctions::abs
    { in x: Rational[1]; return : Rational[1]; }

function '+' specializes RealFunctions::'+'
    { in x: Rational[1]; in y: Rational[0..1]; return : Rational[1]; }
function '-' specializes RealFunctions::'-'
    { in x: Rational[1]; in y: Rational[0..1]; return : Rational[1]; }
function '*' specializes RealFunctions::'*'
    { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function '/' specializes RealFunctions::'/'
    { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function '**' specializes RealFunctions::'**'
    { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function '^' specializes RealFunctions::'^'
    { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }

function '<' specializes RealFunctions::'<'
    { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }
function '>' specializes RealFunctions::'>'
    { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }
function '<=' specializes RealFunctions::'<='
    { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }
function '>=' specializes RealFunctions::'>='
    { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }

function max specializes RealFunctions::max
    { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function min specializes RealFunctions::min
    { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }

function '==' specializes RealFunctions::'=='
    { in x: Rational[0..1]; in y: Rational[0..1]; return : Boolean[1]; }

function gcd
    { in x: Rational[1]; in y: Rational[1]; return : Integer[1]; }

function floor specializes RealFunctions::floor
    { in x: Rational[1]; return : Integer[1]; }
function round specializes RealFunctions::round
    { in x: Rational[1]; return : Integer[1]; }

function ToString specializes RealFunctions::ToString
    { in x: Rational[1]; return : String[1]; }
function ToInteger
    { in x: Rational[1]; return : Integer[1]; }
function ToRational
    { in x: String[1]; return : Rational[1]; }

function sum specializes RealFunctions::sum
    { in collection: Rational[0..*]; return : Rational[1]; }

function product specializes RealFunctions::product
    { in collection: Rational[0..*]; return : Rational[1]; }
```

9.4.11 Integer Functions

9.4.11.1 Integer Functions Overview

This package defines Functions on Integer values, including concrete specializations of the general arithmetic and comparison operations.

9.4.11.2 Elements

```
function abs specializes RationalFunctions::abs
    { in x: Integer[1]; return : Natural[1]; }

function '+' specializes RationalFunctions::'+'
    { in x: Integer[1]; in y: Integer[0..1]; return : Integer[1]; }
function '-' specializes RationalFunctions::'-'
    { in x: Integer[1]; in y: Integer[0..1]; return : Integer[1]; }
function '*' specializes RationalFunctions::'*'
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }
function '/' specializes RationalFunctions::'/'
    { in x: Integer[1]; in y: Integer[1]; return : Rational[1]; }
function '**' specializes RationalFunctions::'**'
    { in x: Integer[1]; in y: Natural[1]; return : Integer[1]; }
function '^' specializes RationalFunctions::'^'
    { in x: Integer[1]; in y: Natural[1]; return : Integer[1]; }
function '%' specializes NumericalFunctions::'%'
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }

function '<' specializes RationalFunctions::'<'
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }
function '>' specializes RationalFunctions::'>'
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }
function '<=' specializes RationalFunctions::'<='
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }
function '>=' specializes RationalFunctions::'>='
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }

function max specializes RationalFunctions::max
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }
function min specializes RationalFunctions::min
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }

function '==' specializes DataFunctions::'=='
    { in x: Integer[0..1]; in y: Integer[0..1]; return : Boolean[1]; }

function '..' specializes ScalarFunctions::'..'
    { in lower: Integer[1]; in upper: Integer[1]; return : Integer[0..*]; }

function ToString specializes RationalFunctions::ToString
    { in x: Integer[1]; return : String[1]; }
function ToNatural
    { in x: Integer[1]; return : Natural[1]; }
function ToInteger
    { in x: String[1]; return : Integer[1]; }

function sum specializes RationalFunctions::sum
    { in collection: Integer[0..*]; return : Integer[1]; }

function product specializes RationalFunctions::product
    { in collection: Integer[0..*]; return : Integer[1]; }
```

9.4.12 Natural Functions

9.4.12.1 Natural Functions Overview

This package defines Functions on Natural values, including concrete specializations of the general arithmetic and comparison operations.

9.4.12.2 Elements

```
function '+' specializes IntegerFunctions::'+'
    { in x: Natural[1]; in y: Natural[0..1]; return : Natural[1]; }
function '*' specializes IntegerFunctions::'*'
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }
function '/' specializes IntegerFunctions::'/'
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }
function '%' specializes IntegerFunctions::'%'
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }

function '<' specializes IntegerFunctions::'<'
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }
function '>' specializes IntegerFunctions::'>'
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }
function '<=' specializes IntegerFunctions::'<='
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }
function '>=' specializes IntegerFunctions::'>='
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }

function max specializes IntegerFunctions::max
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }
function min specializes IntegerFunctions::min
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }

function '==' specializes IntegerFunctions::'=='
    { in x: Natural[0..1]; in y: Natural[0..1]; return : Boolean[1]; }

function ToString specializes IntegerFunctions::ToString
    { in x: Natural[1]; return : String[1]; }
function ToNatural
    { in x: String[1]; return : Natural[1]; }
```

9.4.13 Trig Functions

9.4.13.1 Trig Functions Overview

This package defines basic trigonometric functions on real numbers.

9.4.13.2 Elements

```
feature pi : Real;
inv piPrecision { RealFunctions::round(pi * 1E20) == 314159265358979323846.0 }

function deg {
    in theta_rad : Real[1];
    return : Real[1] = theta_rad * 180 / pi;
}
function rad {
    in theta_deg : Real;
    return : Real[1] = theta_deg * pi / 180;
}

datatype UnitBoundedReal :> Real {
```

```
    inv unitBound { -1.0 <= that & that <= 1.0 }
}

function sin {
    in theta : Real[1];
    return : UnitBoundedReal[1];
}
function cos {
    in theta : Real[1];
    return : UnitBoundedReal[1];
}
function tan {
    in theta : Real[1];
    return : Real = sin(theta) / cos(theta);
}
function cot {
    in theta : Real;
    return : Real = cos(theta) / sin(theta);
}

function arcsin {
    in x : UnitBoundedReal[1];
    return : Real[1];
}
function arccos {
    in x : UnitBoundedReal[1];
    return : Real[1];
}
function arctan {
    in x : Real[1];
    return : Real[1];
}
```

9.4.14 Sequence Functions

9.4.14.1 Sequence Functions Overview

This package defines Functions that operate on general sequences of values. (For Functions that operate on Collection values, see CollectionFunctions.)

9.4.14.2 Elements

```
function '#' specializes BaseFunctions::'#' {
    in seq: Anything[0..*] ordered nonunique;
    in index: Positive[1];
    return : Anything[0..1];
}

function equals{
    in x: Anything[0..*] ordered nonunique;
    in y: Anything[0..*] ordered nonunique;
    return : Boolean[1];
}

function same{
    in x: Anything[0..*] ordered nonunique;
    in y: Anything[0..*] ordered nonunique;
    return : Boolean[1];
}
```

```
function size{
    in seq: Anything[0..*] nonunique;
    return : Natural[1];
}
function isEmpty{
    in seq: Anything[0..*] nonunique;
    return : Boolean[1];
}
function notEmpty{
    in seq: Anything[0..*] nonunique;
    return : Boolean[1];
}
function includes{
    in seq1: Anything[0..*] nonunique;
    in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
}
function includesOnly{
    in seq1: Anything[0..*] nonunique;
    in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
}
function excludes{
    in seq1: Anything[0..*] nonunique;
    in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
}

function union{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function intersection{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function including{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}

function includingAt{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    in index: Positive[1];
    return : Anything[0..*] ordered nonunique;
}

function excluding{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}

function excludingAt{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
```

```
    in startIndex: Positive[1];
    in endIndex: Positive[1] default startIndex;
    return : Anything[0..*] ordered nonunique;
}

function subsequence{
    in seq: Anything[0..*] ordered nonunique;
    in startIndex: Positive[1];
    in endIndex: Positive[1] default size(seq);
    return : Anything[0..*];
}
function head{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..1] = seq[1];
}
function tail{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function last{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..1];
}

behavior add {
    inout seq: Anything[0..*] ordered nonunique;
    in values: Anything[0..*] ordered nonunique;
}
behavior addAt {
    inout seq: Anything[0..*] ordered nonunique;
    in values: Anything[0..*] ordered nonunique;
    in index: Positive[1];
}
behavior remove{
    inout seq: Anything[0..*] ordered nonunique;
    in values: Anything[0..*];
}
behavior removeAt{
    inout seq: Anything[0..*] ordered nonunique;
    in startIndex: Positive[1];
    in endIndex: Positive[1] default startIndex;
}
```

9.4.15 Collection Functions

9.4.15.1 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.

9.4.15.2 Elements

```
function '==' specializes BaseFunctions::'==' {
    in col1: Collection[0..1];
    in col2: Collection[0..1];
    return : Boolean[1];
}

function size {
    in col: Collection[1];
```

```
    return : Natural[1];
}

function isEmpty {
    in col: Collection[1];
    return : Boolean[1];
}

function notEmpty {
    in col: Collection[1];
    return : Boolean[1];
}

function contains {
    in col: Collection[1];
    in values: Anything[*];
    return : Boolean[1];
}

function containsAll {
    in col1: Collection[1];
    in col2: Collection[2];
    return : Boolean[1];
}

function head {
    in col: OrderedCollection[1];
    return : Anything[0..1];
}

function tail {
    in col: OrderedCollection[1];
    return : Anything[0..*] ordered nonunique;
}

function last {
    in col: OrderedCollection[1];
    return : Anything[0..1];
}

function '#' specializes BaseFunctions::'#' {
    in col: OrderedCollection[1];
    in index: Positive[1];
    return : Anything[0..1];
}

function 'array#' specializes BaseFunctions::'#' {
    in arr: Array[1];
    in indexes: Positive[n] ordered nonunique;
    return : Anything[0..1];
    private feature n: Natural[1] = arr.rank;
}
```

9.4.16 Vector Functions

9.4.16.1 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*.

9.4.16.2 Elements

```
abstract function isZeroVector {
    doc
    /*
     * Return whether a VectorValue is a zero vector.
     */

    in v: VectorValue[1];
    return : Boolean[1];
}

abstract function '+' specializes DataFunctions::'+' {
    doc
    /*
     * With two arguments, returns the sum of two VectorValues.
     * With one argument, returns that VectorValue.
     */

    in v: VectorValue[1];
    in w: VectorValue[0..1];
    return u: VectorValue[1];
    inv zeroAddition { w == null or isZeroVector(w) implies u == w }
    inv commutivity { w != null implies u == w + v }
}

abstract function '-' specializes DataFunctions::'-' {
    doc
    /*
     * With two arguments, returns the difference of two VectorValues.
     * With one arguments, returns the inverse
     * of the given VectorValue, that is, the VectorValue that,
     * when added to the original VectorValue, results in
     * the zeroVector.
     */

    in v: VectorValue[1];
    in w: VectorValue[0..1];
    return u: VectorValue[1];
    inv negation { w == null implies isZeroVector(v + u) }
    inv difference { w != null implies v + u == w }
}

abstract function sum0 {
    doc
    /*
     * Return the sum of a collection of VectorValues.
     * If the collection is empty, return a given zero vector.
     */

    in coll: VectorValue[*] nonunique;
    in zero: VectorValue[1];
    inv precondition { isZeroVector(zero) }
```

```
    return s: VectorValue[1] = coll->reduce '+' ?? zero;
}

/* Functions specific to NumericalVectorValues. */

function VectorOf {
    doc
    /*
     * Construct a NumericalVectorValue whose elements are a
     * non-empty list of component NumericalValues.
     * The dimension of the NumericalVectorValue is equal to
     * the number of components.
     */

    in components: NumericalValue[1..*] ordered nonunique;
    return : NumericalVectorValue[1] {
        :>> dimension = size(components);
        :>> elements = components;
    }
}

abstract function scalarVectorMult specializes DataFunctions::'*' {
    doc
    /*
     * Scalar product of a NumericalValue and a NumericalVectorValue.
     */

    in x: NumericalValue[1];
    in v: NumericalVectorValue[1];
    return w: NumericalVectorValue[1];
    inv scaling { norm(w) == x * norm(v) }
    inv zeroLength { isZeroVector(w) implies isZero(norm(w))}
}
alias '*' for scalarVectorMult;

abstract function vectorScalarMult specializes DataFunctions::'*' {
    doc
    /*
     * Scalar product of a NumericalVectorValue and a NumericalValue,
     * which has the same value as the scalar product of the
     * NumericalValue and the NumericalVectorValue.
     */

    in v: NumericalVectorValue[1];
    in x: NumericalValue[1];
    return w: NumericalVectorValue[1] = scalarVectorMult(x, v);
}

abstract function vectorScalarDiv specializes DataFunctions::'/' {
    doc
    /*
     * Scalar quotient of a NumericalVectorValue and a NumericalValue,
     * defined as the scalar product of the inverse of the
     * NumericalValue and the NumericalVectorValue.
     */

    in v: NumericalVectorValue[1];
    in x: NumericalValue[1];
    return w: NumericalVectorValue[1] = scalarVectorMult(1.0 / x, v);
}
```

```
abstract function inner specializes DataFunctions::'*' {
    doc
    /*
     * Inner product of two NumericalVectorValues.
     */

    in v: NumericalVectorValue[1];
    in w: NumericalVectorValue[1];
    return x: NumericalValue[1];
    inv commmutivity { x == inner(w, v) }
    inv zeroInner { isZeroVector(v) or isZeroVector(w) implies isZero(x)}
}

abstract function norm {
    doc
    /*
     * The norm (magnitude) of a NumericalVectorValue, as a NumericalValue.
     */

    in v: NumericalVectorValue[1];
    return l : NumericalValue[1];
    inv squareNorm { l * l == inner(v,v) }
    inv lengthZero { isZero(l) == isZeroVector(v) }
}

abstract function angle {
    doc
    /*
     * The angle between two NumericalVectorValues, as a NumericalValue.
     */

    in v: NumericalVectorValue[1];
    in w: NumericalVectorValue[1];
    return theta: NumericalValue[1];
    inv commutivity { theta == angle(w, v) }
    inv lengthInsensitive { theta == angle(w / norm(w), v / norm(v)) }
}

/* Specialized functions with concrete definitions for CartesianVectorValues. */

function CartesianVectorOf {
    doc
    /*
     * Construct a CartesianVectorValue whose elements are
     * a non-empty list of Real components.
     * The dimension of the NumericalVectorValue is equal
     * to the number of components.
     */

    in components: Real[*] ordered nonunique;
    return : CartesianVectorValue[1] {
        :>> dimension = size(components);
        :>> elements = components;
    }
}
function CartesianThreeVectorOf specializes CartesianVectorOf {
    in components: Real[3] ordered nonunique;
    return : CartesianThreeVectorValue[1];
}
```

```
feature cartesianZeroVector: CartesianVectorValue[3] =
    (
        CartesianVectorOf(0.0),
        CartesianVectorOf((0.0, 0.0)),
        CartesianThreeVectorOf((0.0, 0.0, 0.0))
    ) {
    doc
    /*
     * Cartesian zero vectors of 1, 2 and 3 dimensions.
     */
}
feature cartesian3DZeroVector: CartesianThreeVectorValue[1] =
    cartesianZeroVector[3];

function isCartesianZeroVector specializes isZeroVector {
    doc
    /*
     * A CartesianVectorValue is a zero vector if all its elements are zero.
     */

    in v: CartesianVectorValue[1];
    return : Boolean[1] = v.elements->forAll{in x; x == 0.0};
}

function 'cartesian+' specializes '+' {
    in v: CartesianVectorValue[1];
    in w: CartesianVectorValue[0..1];
    inv precondition { w != null implies v.dimension == w.dimension }
    return u: CartesianVectorValue[1] =
        if w == null? v
        else CartesianVectorOf(
            (1..w.dimension)->collect{in i : Positive; v[i] + w[i]}
        );
}

function 'cartesian-' specializes '-' {
    in v: CartesianVectorValue[1];
    in w: CartesianVectorValue[0..1];
    inv precondition { w != null implies v.dimension == w.dimension }
    return u: CartesianVectorValue[1] =
        CartesianVectorOf(
            if w == null?
                CartesianVectorOf(v.elements->collect{in x : Real; -x})
            else CartesianVectorOf(
                (1..v.dimension)->collect{in i : Positive; v[i] - w[i]}
            )
        );
}

function cartesianScalarVectorMult specializes scalarVectorMult {
    in x: Real[1];
    in v: CartesianVectorValue[1];
    return w: CartesianVectorValue[1] =
        CartesianVectorOf(
            v.elements->collect{in y : Real; x * y}
        );
}
function cartesianVectorScalarMult specializes vectorScalarMult {
    in v: CartesianVectorValue[1];
```

```
    in x: Real[1];
    return w: CartesianVectorValue[1] = cartesianScalarVectorMult(x, v);
}

function cartesianInner specializes inner {
    in v: CartesianVectorValue[1];
    in w : CartesianVectorValue[1];
    inv precondition { v.dimension == w.dimension }
    return x: Real[1] =
        (1..v.dimension)->collect{in i : Positive; v[i] * w[i]}->reduce RealFunctions::'+';
}

function cartesianNorm specializes norm {
    in v: CartesianVectorValue[1];
    return l : NumericalValue[1] = sqrt(cartesianInner(v, v));
}

function cartesianAngle specializes angle {
    in v: CartesianVectorValue[1]; in w: CartesianVectorValue[1];
    inv precondition { v.dimension == w.dimension }
    return theta: Real[1] = arccos(cartesianInner(v, w) / (norm(v) * norm(w)));
}

function sum {
    in coll: CartesianThreeVectorValue[*];
    return : CartesianThreeVectorValue[1] = sum0(coll, cartesian3DZeroVector);
}
```

9.4.17 Control Functions

9.4.17.1 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.

9.4.17.2 Elements

```
abstract function '.' {
    in feature source : Anything[0..*] nonunique {
          abstract feature target : Anything[0..*] nonunique;
    }
    private feature chain chains source.target;
    chain
}

abstract function 'if' {
    in test: Boolean[1];
    in expr thenValue[0..1] { return : Anything[0..*] ordered nonunique; }
    in expr elseValue[0..1] { return : Anything[0..*] ordered nonunique; }
    return : Anything[0..*] ordered nonunique;
}

abstract function '??' {
    in firstValue: Anything[0..*] ordered nonunique;
    in expr secondValue[0..1] { return : Anything[0..*] ordered nonunique; }
    return : Anything[0..*] ordered nonunique;
}

function 'and' {
    in firstValue: Boolean[1];
```

```
    in expr secondValue[0..1] { return : Boolean[1]; }
    return : Boolean[1];
}

function 'or'{
    in firstValue: Boolean[1];
    in expr secondValue[0..1] { return : Boolean[1]; }
    return : Boolean[1];
}

function 'implies'{
    in firstValue: Boolean[1];
    in expr secondValue[0..1] { return : Boolean[1]; }
    return : Boolean[1];
}

abstract function collect {
    in collection: Anything[0..*] ordered nonunique;
    in expr mapper[0..*] {
        in argument: Anything[1];
        return : Anything[0..*] ordered nonunique;
    }
    return : Anything[0..*] ordered nonunique;
}

abstract function select {
    in collection: Anything[0..*] ordered nonunique;
    in expr selector[0..*] {
        in argument: Anything[1];
        return : Boolean[1];
    }
    return : Anything[0..*] ordered nonunique;
}

function selectOne {
    in collection: Anything[0..*] ordered nonunique;
    in expr selector1[0..*] {
        in argument: Anything[1];
        return : Boolean[1]; }
    return : Anything[0..1] =
        collection->select {in x; selector1(x)}[1];
}

abstract function reject{
    in collection: Anything[0..*] ordered nonunique;
    in expr rejector[0..*] {
        in argument: Anything[1];
        return : Boolean[1];
    }
    return : Anything[0..*] ordered nonunique;
}

abstract function reduce {
    in collection: Anything[0..*] ordered nonunique;
    in expr reducer[0..*] {
        in firstArg: Anything[1];
        in secondArg: Anything[1];
        return : Anything[1];
    }
    return : Anything[0..*] ordered nonunique;
```

```
}

abstract function forAll {
    in collection: Anything[0..*] ordered nonunique;
    in expr test[0..*] {
        in argument: Anything[1];
        return : Boolean[1];
    }
    return : Boolean[1];
}

abstract function exists {
    in collection: Anything[0..*] ordered nonunique;
    in expr test[0..*] {
        in argument: Anything[1];
        return : Boolean[1];
    }
    return : Boolean[1];
}

function allTrue {
    in collection: Boolean[0..*];
    return : Boolean[1] = collection->forAll {in x; x};
}

function anyTrue {
    in collection: Boolean[0..*];
    return : Boolean[1] = collection->exists {in x; x};
}

function minimize {
    in collection: ScalarValue[1..*];
    in expr fn[0..*] {
        in argument: ScalarValue[1];
        return : ScalarValue[1];
    }
    return : ScalarValue[1] =
        collection->collect {in x; fn(x)}->reduce min;
}

function maximize {
    in collection: ScalarValue[1..*];
    in expr fn[0..*] {
        in argument: ScalarValue[1];
        return : ScalarValue[1];
    }
    return : ScalarValue =
        collection->collect {in x; fn(x)}->reduce max;
}
```

9.4.18 Occurrence Functions

9.4.18.1 Occurrence Functions Overview

This package defines utility functions that operate on occurrences, primarily related to the time during which those occurrences exist.

9.4.18.2 Elements

```
function '==='  specializes BaseFunctions::'===' {
    doc
    /*
     * Test whether two occurrences are portions of the same life. That is, whether they
     * represent different portions of the same entity (colloquially, whether they have
     * the same "identity").
     */

    in x: Occurrence[0..1];
    in y: Occurrence[0..1];

    return : Boolean[1] = x.portionOfLife == y.portionOfLife;
}

function isDuring {
    doc
    /*
     * Test whether a performance of this function happens during the input occurrence.
     */

    in occ: Occurrence[1];

    private connector all during: HappensDuring[0..1] from self to occ;

    return : Boolean[1] = notEmpty(during);
}

function create {
    doc
    /*
     * Ensure that the start of a given occurrence happens during a performance of this
     * function. The occurrence is also returned from the function.
     */

    inout occ: Occurrence[1];

    private connector : HappensDuring from occ.startShot to self;

    return : Occurrence[1] = occ;
}

function destroy {
    doc
    /*
     * Ensure that the end of a given occurrence happens during a performance of this
     * function. The occurrence is also returned from the function.
     */

    inout occ: Occurrence[0..1];

    private connector : HappensDuring from occ.endShot[0..1] to self;

    return : Occurrence[0..1] = occ;
}

function addNew {
    doc
```

```
    /*
     * Add a newly created occurrence to the given group of occurrences and return the
     * new occurrence.
     */

    inout group: Occurrence[0..*] nonunique;
    inout occ: Occurrence[1];

    private composite step : add {
        inout seq1 = group;
        in seq2 = create(occ);
    }

    return : Occurrence[1] = occ;
}

function addNewAt {
    doc
    /*
     * Add a newly created occurrence to the given ordered group of occurrences at the given
     * index and return the new occurrence.
     */

    inout group: Occurrence[0..*] ordered nonunique;
    inout occ: Occurrence[1];
    in index: Positive[1];

    private composite step : addAt {
        inout seq = group;
        in values = create(occ);
        in startIndex = index;
    }

    return : Occurrence[1] = occ;
}

behavior removeOld {
    doc
    /*
     * Remove a given occurrence from a group of occurrences and destroy it.
     */

    inout group: Occurrence[0..*] nonunique;
    inout occ: Occurrence[0..1];

    private composite step removeStep : remove {
        inout seq = group;
        in values = occ;
    }
    private succession removeStep then destroyStep;
    private composite step destroyStep : destroy {
        inout occ = removeOld::occ;
    }

}

behavior removeOldAt {
    doc
    /*
     * Removes the occurrence at a given index in an ordered group of occurrences
```

```
     * and destroy it.
     */
    inout group: Occurrence[0..*] ordered nonunique;
    in index: Positive[1];

    private feature oldOcc = group[index];

    private composite step removeStep : remove {
        inout seq = group;
        in index = removeOldAt::index;
    }
    private succession removeStep then destroyStep;
    private composite step destroyStep : destroy {
        inout occ = oldOcc;
    }

}
```

10 Model Interchange

10.1 Model Interchange Overview

Model interchange is the capability to interchange models between tools using file-base resources (see Clause 2). The unit of interchange is the *project,* which is defined as follows:

> A project is a set of root namespaces (see 7.2.5.3 and 8.2.3.4.1), including all elements in the ownerhship trees of those namespaces, and a set of references to *used projects,* such that every cross reference from an element in the project is to another element in that project or to an element in one of the used projects.

The root namespaces in a project may be *serialized* into *model interchange files,* using any of the formats given in 10.2. A *project interchange file* is then a compressed archive of model interchange files and additional required metadata, as described in 10.3.

KerML is intended to be used as the basis for building other modeling languages. Project-based model interchange as defined in this clause may also be used to interchange models in such languages. Each of the following subclauses includes descriptions of the allowed adaptations for interchanging models in *KerML-based* languages.

10.2 Model Interchange Formats

A *model interchange file* contains a textual representation (known as a *serialization*) of a single root namespace (see 7.2.5.3 and 8.2.3.4.1) and all the elements in the ownership tree root in that namespace. A model interchange file shall have one 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. A model interchange file in this format shall have the file extension `.kerml`.
2. *JSON*, using a format according to the JSON serialization mapping defined in 10.4. A model interchange file in this format shall have the file extension `.json`.
3. *XML*, using the XML Metadata Interchange [XMI] format based on the MOF-conformant abstract syntax metamodel for KerML. A model interchange file in this format shall have the file extension `.xmi`.

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.

For a KerML-based language:

1. *Textual Notation.* If the language has a textual concrete syntax, then this textual notation may be used as a model interchange file format. The language shall define a distinguishing file extension for files of its textual notation.
2. *JSON.* It shall always be possible to use JSON format as a model interchange file format, using the mapping strategy defined in 10.4, as applied to the abstract syntax of the language.
3. *XML.* If the language is defined using a MOF-conformat abstract syntax, then XMI may be used as a model interchange file format.

A KerML-based-language specification may specify further requirements on what interchange formats must be supported by conforming language tools.

10.3 Model Interchange Projects

A *project interchange file* is contains a single project serialized as a set of model interchange files, archived using the ZIP format [ZIP]. The archive shall contain a model interchange file for each of the root namespaces in the project, each formatted in one of the formats listed in 10.2. In addition, the archive shall contain, at its top level, exactly one file named `.project.json` and exactly one file named `.meta.json`. A KerML project interchange file shall have the file extension `.kpar` (KerML Project Archive).

Other than the use of the file extensions given in 10.2, there are no requirements on the naming of the model interchange files. Nevertheless, they should be named in a way that is compatible across different file systems and that allows for easy reference using International Resource Identifiers (IRIs). The model interchange files may be organized into subdirectories, but this has no impact on the global scope for the project, which is always a flat namespace derived from the root namespaces of the project (see 8.2.3.5). However, each model interchange file shall be identifiable by a unique path in the archive directory structure.

The `.project.json` file shall contain the `InterchangeProject` information shown in Fig. 42, serialized as a single JSON object according to the `Project` schema definition in the `ModelInterchange.json` artifact provided with this specification. Table 12 gives all the properties of the `InterchangeProject` and `InterchangeProjectUsage` elements, consistent with the normative JSON schema. Every element referenced in a model interchange file in a project interchange file shall either also be contained in a model interchange file in that project interchange file, or in one of the projects referenced in the `usage` list for the project interchange file.

The `usage` information for each used project includes an optional `versionConstraint` property. If given, then only versions of the project identified by the `resource` property that meet this constraint may be used. For an interchanged project, the version is as given in its `version` property. It is recommended, but not required, that *semantic versioning* (see https://semver.org/) be used for the version numbering of interchange projects and *semantic versioning ranges* (see, e.g., https://docs.npmjs.com/cli/v6/using-npm/semver#ranges) be used for version constraints. Tools that support such version formatting should report any version constraint violations when importing an interchange project, for any used projects with dereferencable `resource` IRIs.

The `.meta.json` file shall contain further metadata on the project interchange file, serialized as a single JSON object according to the Meta schema definition in the `ModelInterchange.json` artifact provided with this specification. Table 13 describes all the fields specified in the normative JSON schema.

A project interchange file for a KerML-based language shall include model interchange files specific to that language (as described in 10.2). Such a project interchange file may use the generic `.kpar` extension, or it may define its own language-specific extension. If it uses the `.kpar` extension, then the metadata for the file shall identify the KerML-based language metamodel (see Table 13). Each project interchange file shall only contain models in a single language, but it shall be able to have used projects both in the same language and in KerML (such as from the Kernel Model Libraries). A KerML-based-language specification may also allow for project interchange files that use projects in other KerML-based languages.

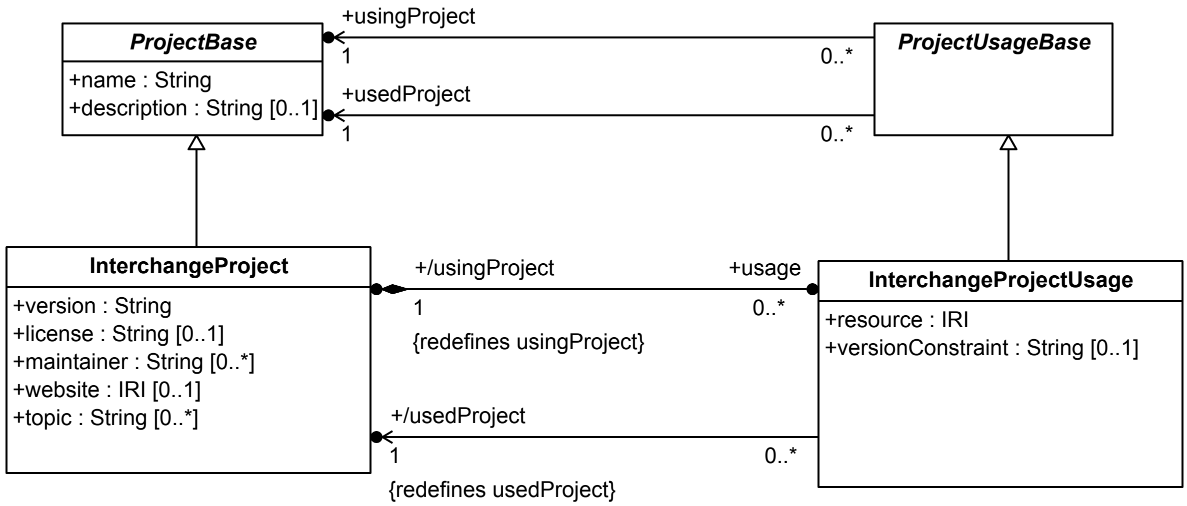

Figure 42. Interchange Projects

Table 12. Interchange Project Information

Property	Type	Mandatory	Description
`name`	string	yes	The name of the project.
`description`	string	no	A description of the project.
`version`	string	yes	The version of the project being interchanged.
`license`	string	no	The license by which project content may be used.
`maintainer`	array (of strings)	no	A list of names of maintainers of the project.
`website`	IRI	no	An IRI for a Web site with further information on the project.
`topic`	array (of strings)	no	A list of topics relevant to the project.
`usage`	array (of objects)	no	A list of project usage entries, one for each project used by the project being interchanged, with properties as given below.
`resource`	IRI	yes (within a `usage`)	An IRI identifying the project being used. If the IRI is dereferenceable, it should resolve to a project interchange file for the used project.

Property	Type	Mandatory	Description
`versionConstraint`	string	no (within a `usage`)	A constraint on the allowable versions of a used project.

Table 13. Interchange Project Metadata

Name	Type	Mandatory	Description
`index`	object	yes	An index of the global scope of the project, specified as a JSON object with a key for each name, whose associated value is the path to the model interchange file containing the root namespace for the named element. (See Notes 1 and 2.)
`created`	string	yes	The date and time of the creation of the project interchange file, in ISO 8601 format [ISO8601].
`metamodel`	IRI	no	An IRI identifying the metamodel of the modeling language of the models being interchanged in this project interchange file. (See Note 3.)
`includesDerived`	Boolean	no	Whether derived property values are included in the model interchange files. (See Note 4.)
`includesImplied`	Boolean	no	Whether implied relationships are included in the model interchange files. (See Note 5.)
`checksum`	object	no	A dictionary mapping paths to some or all of the model interchange files to a list of one or more objects with the two properties given below. (See Note 2.)
`value`	string	yes (within a `checksum`)	The checksum computed according to the checksum `algorithm`.

Name	Type	Mandatory	Description
algorithm	string	yes (within a checksum)	Identification of the algorithm used to computed the checksum value. (See Note 6.)

Notes

1. The index cross-references all the non-null shortNames and names of all the top-level elements of the root namespaces of the project (see 7.2.5.3 and 8.2.3.5) to the model interchange file of the root namespace that contains the element. Note that, while the names of all top-level elements in a root namespace must be unique, it is allowable (though not recommended) for top-level elements in different root namespaces of a project to have the same name.
2. File paths are always relative to the root of the project interchange file archive, with path segments separated by the forward slash symbol /, ending in a file name with extension (e.g., structure/assembly/Body.json)
3. For an OMG-standardized language, metamodel shall be the version-specific URI specified by OMG to identify the language. For KerML, this URI has the form https://www.omg.org/spec/KerML/yyyymmxx, with a version-specific date stamp "yyyymmxx". If metamodel is not given, the default is KerML (for a project interchange file with the .kpar extension).
4. If includesDerived = true, then the serializations in all XMI and JSON format model interchange files in the project interchange file shall include values for all derived properties. If includesDerived = false, then XMI and JSON formatted model interchanges files shall not include values for any derived properties. If includesDerived is not given, then whether derived property values are included may vary one model interchange file to another, and it is also allowable for some values to be included for some derived properties and not others.
5. If includesImplied = true, then the serializations in all XMI and JSON formatted model interchange files in the project interchange file shall include all implied relationships, and the isImpliedIncluded property shall have the value true for all elements (see 8.3.2.1 on isImpliedIncluded). If includesImplied = false, then XMI and JSON formatted model interchange files shall not include any implied relationships, and the isImpliedIncluded property shall have the value false for all elements. If includesImplied is not given, then whether implied relationships are included may vary from one model interchange file to another, and from element to element, as recorded by the value of the includesImplied property for each element.
6. Valid values for the checksum algorithm are
   - SHA1, SHA224, SHA256, SHA-384, SHA3-256, SHA3-384, SHA3-512 [SHS]
   - BLAKE2b-256, BLAKE2b-384, BLAKE2b-512, BLAKE3 [BLAKE]
   - MD2, MD4, MD5, MD6 [MD]
   - ADLER32 [ADLER]

10.4 JSON Serialization

10.4.1 Serialization Overview

The JSON serialization format can be used to interchange any model conformant with the KerML abstract syntax. Each root namespace shall correspond with a model interchange file with the file extension .json and contain serializations of all model elements in the ownership tree root in that namespace. The contents of this file shall be in the JSON (JavaScript Object Notation) format [JSON] and, for KerML, conform to the JSON schema definitions in the KerML.json artifact provided with this specification. Other KerML-based languages may extend this schema or define their own schema, consistent with the serialization strategy defined here as applied to the abstract syntax of those languages.

The following subclauses describe the serialization strategy, as realized in the normative JSON schema for KerML.

10.4.2 Primitive Type Serialization

The UML primitive types used in the KerML abstract syntax map directly to core JSON Schema types, as shown in Table 14.

Table 14. UML Primitive Type Serialization

UML Primitive Type	JSON Schema Type
Boolean	boolean
Integer	integer
Real	number
String	string

10.4.3 Enumeration Serialization

Enumeration values map to a JSON Schema `string` with a value that is the `name` of the enumeration literal, with the same capitalization as defined for the literal in the abstract syntax. For example, `VisibilityKind::public` maps to the string `"public"`.

10.4.4 Element Reference Serialization

Values of abstract syntax properties typed by a metaclass (that is, `Element` or one of its subclasses) map to a JSON Schema `object` with a single field `@id`. The value of `@id` is a JSON Schema `string` with a value equal to the value of the `elementId` of the `Element`. For example:

```
{
  "@id": "15fe7607-ceb8-38bb-bd04-dde8ca657cec"
}
```

10.4.5 Element Serialization

A model element maps to a JSON Schema `object` with fields `@id`, `@type`, and a set of its attributes. The field `@id` has a `string` value equal to the value of `Element::elementId`. The field `@type` has a `string` value equal to the `name` of the specific MOF type of the element, e.g. `"Structure"`.

The remaining JSON Schema fields are mapped from the set of MOF properties specified as attributes of the MOF type of the element. This shall include all owned and inherited properties. In addition, while redefined properties are *not* inherited under MOF/UML rules, they *shall* be included in the set of properties serialized for the element if they have a different name than the redefining property.

Each of these maps to a JSON Schema field, where the name of the field is equal to the name of the attribute and the value is equal to the serialization of the attribute value as described in the preceding subclauses. The value must adhere to the allowed multiplicity of the MOF attribute:

- A multiplicity of `[1..1]` requires a non-null value.
- A multiplicity of `[0..1]` allows a value or null
- A multiplicity with an upper bound greater than 1 maps to a JSON Schema `array` with values equal to the serialization of the attribute values described in the preceding subclauses.

10.4.6 Model Serialization

A root namespace maps to a JSON Schema `array` with values equal to the serialization, as described in the preceding subclauses, of all model elements in the ownership tree rooted in that namespace.

A Annex: Model Execution

(Informative)

A.1 Overview

The language semantics in this specification give conditions to check whether classifiers have been instantiated properly (see 7.3.2.1). For structures this includes their parts and other required objects, as well as feature values and links between them. For behaviors, this includes their steps and other required performances, as well as timing links between them. These two kinds of classifiers are typically interrelated, structures can require behaviors for proper instantiation and vice-versa.

This annex outlines a procedure for incrementally instantiating (executing) classifiers to ensure the completed instances will pass the check above (satisfy classifier conditions). The order of instantiation obeys any timing specified by the classifier. For example, some structures might require others to exist first, such as parents before their children, or parts of a car before assembly, while behaviors typically require some steps to happen only after others finish, such as painting objects before drying them. It covers the basic patterns needed to aid development of a complete execution procedure.

A.2 Modelling Instances and Feature Values

Instances in this annex are modeled in KerML, rather than as runtime data structures. Execution is taken to be creating these modeled instances in an order specified by their classifiers.

Instances are also modeled as classifiers (called *atoms* in this annex) that each correspond to their own single (runtime) instance. Atoms are all disjoint from each other, but not necessarily from other (non-atom) classifiers (such as the ones being instantiated). In the example below, MyBike and YourBike are atoms. They are both classified by Bicycle (and Vehicle by specialization).

```
classifier Vehicle;
classifier Bicycle specializes Vehicle;
classifier MyBike [1] specializes Bicycle;
classifier YourBike [1] specializes Bicycle disjoint from MyBike;
```

Atoms in this annex are indicated by a user-defined keyword before classifier definitions, with multiplicity and disjointness implied by the keyword, as below.

```
classifier Atom;
metaclass <atom> AtomMetadata specializes Metaobject {
    baseType = Atom meta KerML::Classifier;
}

classifier Vehicle;
classifier Bicycle specializes Vehicle;

#atom
classifier MyBike specializes Bicycle;
#atom
classifier YourBike specializes Bicycle;
```

Atoms are assigned as feature values by typing a feature with them, or a union of atoms, and restricting the feature multiplicity as needed to match the number of atoms being assigned. The example below creates a classifier for the bicycle atoms above (OurBicycle), then redefines a feature (`stores`) to be typed by it. The multiplicity is restricted to the exact number of atoms creating during execution (2).

```
classifier Garage {
    feature stores : Bicycle [*];
}
classifier OurBicycle unions MyBike, YourBike;

#atom
classifier OurGarage specializes Garage {
    feature redefines stores : OurBicycle [2];
}
```

A.3 Instantiation Procedure

A.3.1 Overview

The instantiation procedure is described in cases of increasing capability. A.3.2 through A.3.4 cover features, including connectors, without any timing specified. These are applicable to structure and behavior, though the examples are structural. The rest of the procedure adds timing, first for structures, then behaviors.

A.3.2 Without connectors

Take the example below to illustrate the procedure, a (non-association) classifier without connectors (features typed by associations).

```
classifier Bicycle {
    feature rollsOn : Wheel [2];
    feature holdsWheel : BikeFork [*];
}
classifier Wheel;
classifier BikeFork;
```

The instantiation procedure starts with

1. Create an atom of the classifier being instantiated (Bicycle).
2. Identify features of the instantiated classifier with lower multiplicity greater than zero that are not connectors or other features typed by associations (`rollsOn`).
3. Create atoms for the types of the above features (Wheel), at least up to the lower multiplicity of each feature (2), and assign them as values of the feature.

The model being executed in this example does not specify timing, though it is typically expected that

- classifiers are produced before their atoms.
- Atoms are produced before they are assigned as values or otherwise used by another atom.

The first instantiation step produces the first atom below

```
#atom
classifier MyBike specializes Bicycle;
```

The third creates the rest and modifies the one above. Atoms appear each time they are modified (MyBike), to highlight execution order.

```
#atom
classifier MyWheel1 specializes Wheel;
#atom
classifier MyWheel2 specializes Wheel;
```

```
classifier MyWheel unions MyWheel1, MyWheel2;

#atom
classifier MyBike specializes Bicycle {
    feature redefines rollsOn : MyWheel;
}
```

A.3.3 One-to-one connectors

This covers connectors that

- have multiplicity 1 at both ends, but unrestricted (`*`) overall.
- are not timing or binding connectors.

The first above requires the connected features to have the same number of values. When this is not possible, such as the multiplicities of the connected features being incompatible (do not overlap, as in 0..1 and 2..*), the classifier is not instantiable (satisfiable).

The example below adds a connector to the example in *Without Connectors*, with end multiplicities requiring each wheel to be fixed to its own fork, and vice-versa.

```
classifier Bicycle {
    feature rollsOn : Wheel [2];
    feature holdsWheel : BikeFork [*];
    connector fixWheel : BikeWheelFixed from rollsOn [1] to holdsWheel [1];
}
assoc BikeWheelFixed {
    end feature wheel : Wheel;
    end feature fixedTo : BikeFork;
}
```

The instantiation procedure from *Without connectors* continues with

4. Identify connectors of the classifier being instantiated (`fixWheel`).
5. For each connector above
    a. Create association atoms for the types of the connectors identified in step 4 (BikeWheelFixed). See below for how many.
    b. Assign the two participant (end) features (`wheel` and `fixed`) in each association atom, with values taken from the corresponding connected feature (`rollsOn` and `holdsWheel`). See below for which values are taken.
    c. Assign the association atoms above as values of the corresponding connectors.

   For end multiplicity 1 on both ends
    - Create the same number of association atoms as there are values of the connected features with the most values at the time the association atoms are created (2).
    - Assign each connected feature value as participant in exactly one assocation atom.
      If one connected feature has fewer values than the other, create atoms for the type of that feature (`holdsWheel`) up to the number in the other feature (`rollsOn`, 2), and assign them as values of the feature with fewer values.

The model being executed in this example does not specify timing, though it is typically expected that

- association atoms are created just after values are assigned to connected features, whereupon the instantiation steps above could be taken on each connector right after its connected features are assigned values during step 3, see Clause .

After the instantiations in *Without connectors*, the steps above produce the following atoms. First, 5.a creates as many association atoms for the connector (`fixWheel`) as the connected feature with the most values (2 in `rollsOn`, assigned in *Without connectors*).

```
#atom
assoc MyBikeWheel1_Fork1_BWF_Link specializes BikeWheelFixed;
#atom
assoc MyBikeWheel2_Fork2_BWF_Link specializes BikeWheelFixed;
```

Before 5.b assigns participant features, the one-to-one connector end multiplities require additional atoms for the connected feature with fewer values (`holdsWheel`), to match the number of the values of the other connected feature (`rollsOn`).

```
#atom
classifier MyBikeFork1 specializes BikeFork;
#atom
classifier MyBikeFork2 specializes BikeFork;

classifier MyBikeFork unions MyBikeFork1, MyBikeFork2;

#atom
classifier MyBike specializes Bicycle {
    feature redefines rollsOn : MyWheel;
    feature redefines holdsWheel : MyBikeFork;
}
```

Then 5.b assigns participant feature values to the association atoms created in 5.a, choosing in this execution to fix the first and second wheels to the first and second forks, respectively,

```
#atom
assoc MyBikeWheel1_Fork1_BWF_Link specializes BikeWheelFixed {
    end feature redefines wheel : MyWheel1;
    end feature redefines fixedTo : MyBikeFork1;
}
#atom
assoc MyBikeWheel2_Fork2_BWF_Link specializes BikeWheelFixed {
    end feature redefines wheel : MyWheel2;
    end feature redefines fixedTo : MyBikeFork2;
}
```

Finally 5.c assigns the association atoms to the connector.

```
classifier MyBikeWheel_Fork_BWF_Link
    unions MyBikeWheel1_Fork1_BWF_Link, MyBikeWheel2_Fork2_BWF_Link;
#atom
classifier MyBike specializes Bicycle {
    feature redefines rollsOn : MyWheel;
    feature redefines holdsWheel : MyBikeFork;
    connector redefines fixWheel : MyBikeWheel-Fork-BWF-Link [2]
        from rollsOn [1] to holdsWheel [1];
}
```

A.3.4 One-to-unrestricted connectors

This covers connectors that

- have multiplicity 1 at one end and unrestricted (`*`) at the other, and unrestricted overall.

- are not timing or binding connectors.

The first above enables the feature connected at the unrestricted end to have any number of values in satisfiable models, but if it has any values, all those values must be linked to exactly one (not necessarily unique) value of the other connected feature (at the multiplicity 1 end) in the same instance of the classifier being instantiated. When this is not possible, for example due to the connector's association multiplicities being too restrictive (such as not allowing for links to all the values of the connected features), the classifier is not instantiable (satisfiable).

The example below adds another feature and connector to the example in A.3.3. Every basket is intended to be fixed to one of the forks, though more than one basket might be fixed to the same fork, or some forks might have no baskets, ot there might be no baskets at all.

```
classifier Bicycle {
    feature carrier : BikeBasket [*];
    connector carrierFixed : BikeBasketFixed from carrier [*] to holdsWheel [1];
}
classifier BikeBasket;

assoc BikeBasketFixed {
    end feature basket : BikeBasket;
    end feature fixedTo : BikeFork;
}
```

Then the instantiation procedure from A.3.3 is amended for one unrestricted end:

5. For each connector above
    a. .., b. ..., c. ..., ...

   For end multiplicity 1 on one end and unrestricted on the other
    - Create the same number of association atoms as there are values of the connected feature at the unrestricted end (`carrier`) at the time the association atoms are created.
    - Assign each connected feature value at the unrestricted end as participant in exactly one assocation atom.

Instantiation proceeds as in A.3.3, modifying the classifier atom created there (MyBike), except the number of association atoms created is determined by the number of values of the connected feature at the unrestricted end (`carrier`), choosing in this execution to fix two baskets to the same fork.

```
#atom
classifier MyBikeBasket1 specializes BikeBasket;
#atom
classifier MyBikeBasket2 specializes BikeBasket;

classifier MyBikeBasket unions MyBikeBasket1, MyBikeBasket2;

#atom
classifier MyBike specializes Bicycle {
  feature redefines carrier : MyBikeBasket [2];
}
#atom
assoc MyBikeBasket1_Fork1_BBF_Link specializes BikeBasketFixed {
    end feature redefines basket : MyBikeBasket1;
    end feature redefines fixedTo : MyBikeFork1;
}
#atom
assoc MyBasket2_BikeFork1_BBF_Link specializes BikeBasketFixed {
    end feature redefines basket : MyBikeBasket2;
    end feature redefines fixedTo : MyBikeFork1;
```

```
}

classifier MyBikeBasket_Fork_BBF_Link
    unions MyBikeFork1_Basket1_BBF_Link, MyBikeFork1_Basket2_BBF_Link;

#atom
classifier MyBike specializes Bicycle {
    feature redefines carrier : MyBikeBasket [2];
    connector redefines carrierFixed : MyBikeBasket_Fork_BBF_Link [2]
        from carrier [*] to holdsWheel [1];
}
```

A.3.5 Timing for structures

Classes are classifiers for things that exist in time (*occurrences*), as compared to numbers or other mathematical entities. It usually matters when these things come into and go out of existence, at least relative to each other. For example, in structures it is typically intended that parts exist for at least as long as the thing they are part of. In the bicycle example above, the wheel atoms should exist at least as long as the bicycle atom. A simple way to do this is for a structure and its parts to all exist at exactly the same time, as show below. Bicycle is class with its part features subset from *`timeCoincidentOccurrences`*, ensuring the values of `rollsOn` and `holdsWheel` happen (start and end) at exactly the same time as the bicycle occurrence they are part of, see 9.2.4.1.

```
class Bicycle specializes Occurrence {
    feature rollsOn : Wheel [2] subsets timeCoincidentOccurrences;
    feature holdsWheel : BikeFork [2] subsets timeCoincidentOccurrences;
}
```

The keyword **`struct`** implies specialization from *Occurrence* and highlights the structural application of classes. With it the example above becomes

```
struct Bicycle {
    feature rollsOn : Wheel [2] subsets timeCoincidentOccurrences;
    feature holdsWheel : BikeFork [2] subsets timeCoincidentOccurrences;
}
```

Some features always include the thing featuring them as a value (logically *reflexive*), such as *`self`* (see 9.2.2.1), which subsets *`timeCoincidentOccurrences`*, because occurrences always exist at the same time as themselves (see 7.3.4.4 about feature subsetting). The instantiation procedure in A.3.2 is amended below assign values to reflexive features when it creates atoms for features with lower multiplicity greater than zero (3a), as well as assign values to features being subsetted (3b).

3. ...
    a. For features that always have their featuring thing as a value (at least *`self`* and the features it subsets), assign that atom as a value.
    b. If a feature subsets others (`rollsOn` and `holdsWheel` subset *`timeCoincidentOccurrences`*, which subsets *`self`*), assign values to the others also.

The additional instantiation steps produce the atom below by

- Assigning MyBike with itself for *`self`*.
- Introducing a class for all part atoms of MyBike, as well as the assembled MyBike, and assigning *`timeCoincidentOccurrences`* with it.

The model being executed requires all the occurrences to come into and go out of existence at the same time, but since this is not possible when sequentially creating atoms, the ones below appear before they are assigned as values or otherwise used by another atom.

```
struct MyBikeTimeCoincident unions MyWheel, MyBikeFork, MyBike;

#atom
struct MyBike specializes Bicycle {
    feature redefines self : MyBike;
    feature redefines timeCoincidentOccurrences : MyBikeTimeCoincident [5];
    feature redefines rollsOn : MyWheel;
    feature redefines holdsWheel : MyBikeFork;
}
```

It is more realistic for parts to exist before the things they they are a part of, such as the wheels and forks above existing before they are assembled into a bicycle. It also might be that the parts outlive the bicycle, if it's only disassembled rather than completely destroyed. A simple way to do this is for parts to exist longer than their assembly, as specified below by a *HappensDuring* connector (see 9.2.4.1) linking the bicycle (*`self`*) to all its parts (`allParts`), specified as a union of the part features (equivalent to part features subsetting the union feature, but excluding values that are not in the part features). This ensures each bicycle exists during the time their parts do, but enables the parts to exist before assembly into a bicycle, and after disassembly, when the bicycle no longer exists. They still might all exist at the exactly the same time, as in the previous example, because things that exist at the same time all happen during each other.

```
struct Bicycle {
    feature rollsOn : Wheel [2];
    feature holdsWheel : BikeFork [2];
    feature allParts : Occurrence unions rollsOn, holdsWheel;
    connector b_during_ap : HappensDuring from self [1] to allParts [*];
}
```

The instantiation procedure above is amended again to assign values to union features.

3. ...
    a. ..., b ...
    c. If a feature unions others (`allParts` unions `rollsOn` and `holdsWheel`), treat the others as subsetting the union feature, but only assigning values that are in the subsets.

The instantiation procedure in A.3.4, with the amendments above to A.3.2, produces the additional or modified atoms below.

```
#atom
assoc MyBike_During_Wheel1_Link specializes HappensDuring {
    end feature redefines shorterOccurrence : MyBike;
    end feature redefines longerOccurrence : MyWheel1;
}
#atom
assoc MyBike_During_Wheel2_Link specializes HappensDuring {
    end feature redefines shorterOccurrence : MyBike;
    end feature redefines longerOccurrence : MyWheel2;
}
#atom
assoc MyBike_During_Fork1_Link specializes HappensDuring {
    end feature redefines shorterOccurrence : MyBike;
    end feature redefines longerOccurrence : MyBikeFork1;
}
#atom
assoc MyBike_During_Fork2_Link specializes HappensDuring {
    end feature redefines shorterOccurrence : MyBike;
    end feature redefines longerOccurrence : MyBikeFork2;
}
```

```
assoc MyBike_During_Parts_Link  specializes HappensDuring
    unions MyBike_During_Wheel1_Link, MyBike_During_Fork1_Link,
           MyBike_During_Wheel2_Link, MyBike_During_Fork2_Link;

struct MyBikeParts unions MyWheel, MyBikeFork;

#atom
struct MyBike specializes Bicycle {
    feature redefines rollsOn : MyWheel;
    feature redefines holdsWheel : MyBikeFork;
    feature redefines allParts : MyBikeParts [4];

    feature redefines self : MyBike;
    connector redefines b_during_ap : MyBike_During_Parts_Link [4]
        from self [1] to allParts [*];
}
```

Parts are sometimes expected not to exist after their structures, such as when a bicycle is completely destroyed, rather than just disassembled. The wheels and forks above would not exist after the bicycle they are part of, even though they might have existed before it (was assembled). Since parts can be replaced over time, the only ones destroyed are those in the bicycle at the time it is destroyed (the parts replaced earlier are not affected because they are no longer in the bicycle). This is specified below by a *HappensWhile* connector, equivalent to a *HappenDuring* connector in both directions, ensuring the ends (*`endShot`*) of the parts (at the time the bicycle ends) happen at the same time as the bicycle (the connected feature values are *`timeCoincidentOccurrences`* of each other). End shots are instantaneous occurrences that happen at the end of another occurrence (*life*), but represent the same thing as that occurrence, see 9.2.4.1. The ends of the parts are identified by "navigating" (*chaining*) through a series of features, each providing a value on which to get the next value in the navigation, starting with the end of the bicycle, see 7.3.4.6.

```
struct Bicycle {
    ...
    feature redefines endShot : Bicycle;
    connector be_while_pe : HappensWhile
        from endShot [1] to endShot.allParts.endShot[*];
}
```

The keyword **`composite`** implies the end timing above when used in defining `rollsOn` and `holdsWheel`, as below.

```
struct Bicycle {
    composite feature rollsOn : Wheel [2];
    composite feature holdsWheel : BikeFork [2];
    ...
}
```

The instantiation steps in A.3.2 will assign a value to *`endShot`*, because it has multiplicity [1], but this is delayed until everything else required in the structure has occurred, due to mandatory *HappensBefore* connectors to *`endShot`*. With the amendments above, the steps in A.3.4 produce the additional or modified atoms below.

```
  /* End atoms */
#atom
struct MyWheel1End specializes Wheel;
#atom
struct MyWheel1 specializes Wheel {
    feature redefines endShot : MyWheel1End;
}
```

```
#atom
struct MyWheel2End specializes Wheel;
#atom
struct MyWheel2 specializes Wheel {
    feature redefines endShot : MyWheel2End;
}
struct MyBikeFork1End specializes BikeFork;
#atom
struct MyBikeFork1 specializes BikeFork {
    feature redefines endShot : MyBikeFork1End;
}
struct MyBikeFork2End specializes BikeFork;
#atom
struct MyBikeFork2 specializes BikeFork {
    feature redefines endShot : MyBikeFork2End;
}
#atom
struct MyBikeEnd specializes Bicycle;

  /* HappensWhile atoms */
#atom
assoc MyBikeEnd_While_Wheel1End_Link specializes HappensWhile {
    end feature redefines thisOccurrence : MyBikeEnd;
    end feature redefines thatOccurrence : MyWheel1End;
}
#atom
assoc MyBikeEnd_While_Wheel2End_Link specializes HappensWhile {
    end feature redefines thisOccurrence : MyBikeEnd;
    end feature redefines thatOccurrence : MyWheel2End;
}
#atom
assoc MyBikeEnd_While_Fork1End_Link specializes HappensWhile {
    end feature redefines thisOccurrence : MyBikeEnd;
    end feature redefines thatOccurrence : MyBikeFork1End;
}
#atom
assoc MyBikeEnd_While_Fork2End_Link specializes HappensWhile {
    end feature redefines thisOccurrence : MyBikeEnd;
    end feature redefines thatOccurrence : MyBikeFork2End;
}

assoc MyBikeEnd_While_PartsEnd_Link  specializes HappensWhile
    unions MyBikeEnd_While_Wheel1End_Link, MyBikeEnd_While_Fork1End_Link,
           MyBikeEnd_While_Wheel2End_Link, MyBike_While_Fork2End_Link;

#atom
struct MyBike specializes Bicycle {
    ...
    feature redefines endShot : MyBikeEnd;
    connector redefines be_while_pe : MyBikeEnd_While_PartsEnd_Link [4]
        from endShot [1] to endShot.allParts.endShot[*];
}
```

A.3.6 Timing for behaviors, Sequences

Behaviors also exist in time, but they come into and go out of existence differently than structures, relative to each other. For example, in behaviors it is typically intended that its steps happen

- during the behavior occurrence they are part of, but not last as long as it does.

- before or after other steps in the same behavior occurrence.

These can be modelled by

- subsetting step features (those typed by behaviors) from *timeEnclosedOccurrences*, ensuring they happen during the occurrence they are a step of, but are not required to happen the entire time, as with *timeCoincidentOccurrences*.
- linking steps by *HappensBefore* connectors to specify the order they should occur in.

The example below does this for a behavior with three steps. Only the first one (`paint`) is required (by its multiplicity), indicating the behavior starts there, while the rest (`dry` and `ship`) are unrestricted, to prevent the instantiation procedure from giving values to (performing) them too early. This is left to the end multiplicities of the timing connectors (`p_before_d` and `d_before_s`), which require their later step to happen once each time the earlier step does, and vice-versa, see A.3.3.

```
class Manufacture specializes Occurrence {
    feature paint : Paint [1] subsets timeEnclosedOccurrences;
    feature dry : Dry [*] subsets timeEnclosedOccurrences;
    connector p_before_d: HappensBefore from paint [1] to dry [1];
    feature ship : Ship [*] subsets timeEnclosedOccurrences;
    connector d_before_s: HappensBefore from dry [1] to ship [1];
}
behavior Paint;
behavior Dry;
behavior Ship;
```

The keyword

- **`step`** implies subsetting `timeEnclosedOccurrences`. These are features of behaviors typed by behaviors.
- **`succession`** implies typing connectors by *HappensBefore*, and indicates the ends with `first` and `then`, instead of `from` and `to`, respectively.

With these the example above becomes

```
behavior Manufacture {
    step paint : Paint [1];
    step dry : Dry [*];
    succession p_before_d first paint [1] then dry [1];
    step ship : Ship [*];
    succession d_before_s first dry [1] then ship [1];
}
```

The instantiation procedure in A.3.3 produces the atoms below, including

- creating step atoms required by one-to-one connectors, and
- the typically expected order to create associaton atoms (association atoms are created and assigned just after values are assigned to connected features), taken as required for behavioral connectors.

In this example, it results in creating the atoms below ("taking" steps) in the order typically expected for behavioral execution, even though the procedure is the same as for structural execution. Atoms appear below each time they are modified (MyManufacture), to highlight this.

It starts by creating and assigning an atom for `paint`, to satisfy its multiplicity, see A.3.2. The procedure ignores the others because their multiplicities do not require any values.

```
#atom
behavior MyManufacture specializes Manufacture;
#atom
behavior MyPaint specializes Paint;
#atom
behavior MyManufacture specializes Manufacture {
    feature redefines timeEnclosedOccurrences : MyPaint [1];
    step redefines paint : MyPaint;
}
```

Step 5 in A.3.3 reacts to the new step atom above by looking for connectors from it with multiplicity [1] at the opposite end, finding `p_before_d`, then creating and assigning an atom for the step connected at that end (`dry`), to satisfy that end's multiplicity.

```
#atom
behavior MyDry specializes Dry;

#atom
assoc MyPaint_Before_Dry_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyPaint;
    end feature redefines laterOccurrence : MyDry;
}

behavior MyManufactureStepsPD unions MyPaint, MyDry;

#atom
behavior MyManufacture specializes Manufacture {
    feature redefines timeEnclosedOccurrences : MyManufactureStepsPD [2];
    step redefines paint : MyPaint;
    step redefines dry : MyDry [1];
    succession redefines p_before_d : MyPaint_Before_Dry_Link [1]
        first paint then dry;
}
```

Step 5 repeats for the remaining connector and step (`d_before_s` and `ship`).

```
#atom
behavior MyShip specializes Ship;

#atom
assoc MyDry_Before_Ship_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyDry;
    end feature redefines laterOccurrence : MyShip;
}

behavior MyManufactureStepsPDS unions MyManufactureStepsPD, MyShip;

#atom
behavior MyManufacture specializes Manufacture {
    feature redefines timeEnclosedOccurrences : MyManufactureStepsPDS [3];
    step redefines paint : MyPaint;
    step redefines dry : MyDry [1];
    succession redefines p_before_d : MyPaint_Before_Dry_Link [1] first paint then dry;
    step redefines dry : MyShip [1];
    succession redefines d_before_s : MyDry_Before_Ship_Link [1] first dry then ship;
}
```

A.3.7 Timing for behaviors, Decisions and merges

Decisions and merges are steps that enable sequences to be selected during execution, rather than ahead of time in models, as in A.3.6:

- Decisions are steps with multiple outgoing successions, but only one is traversed during each execution of the decision.
- Merges are steps with multiple incoming successions, but only one is traversed during each execution of the merge.

These are modelled by

- Optional connector end multiplicities ([0..1]) on ends of the outgoing and incoming successions opposite decision and merge steps, respectively. This enables execution to determine which succession is traversed for each decision and merge.
- Decision and merges steps typed by *DecisionPerformance* and *MergePerformance* from the Kernel library, respectively. These enable additional timing constraints that require exactly one succession to be traversed for each decision and merge.

The example below includes a decision and merge, which appear in sequence like other steps, except with optional branching successions out and in, respectively. The timing constraints at the end ensure successions going out of the decision step and coming into the merge step have exactly one value (*HappensBefore* link) for each time those steps happen, identified by the library features `outgoingHBLink` and `incomingHBLink` of *DecisionPerformance* and *MergePerformance*, respectively, which are required to have exactly one value for each performance.

```
behavior Manufacture {
      /* Before decision. */
    step admit : Admit [1];
    succession a_before_i first admit [1] then inspect [1];

      /* Decision. */
    step inspect : DecisionPerformance [*];

      /* Two decision branches. */
    succession i_before_f first inspect [1] then finish [0..1];
    step finish : Touchup [*];
    succession i_before_r first inspect [1] then recycle [0..1];
    step recycle : MarkForRecycling [*];

      /* Two merge branches. */
    succession f_before_ms first finish [0..1] then mShip [1];
    succession r_before_ms first recycle [0..1] then mShip [1];

      /* Merge */
    step mShip : MergePerformance [*];

      /* After merge */
    succession ms_before_s first mShip [1] then ship [1];
    step ship : Ship [*];

      /* Decision and merge timing constraints. */
    feature inspectOutgoingHBLinks : HappensBefore [*] unions i_before_f, i_before_r;
    connector bindIOHBL : SelfLink
        from inspectOutgoingHBLinks [1] to inspect.outgoingHBLink [1];
    feature mShipIncomingHBLinks : HappensBefore [*] unions f_before_ms, r_before_ms;
    connector bindmSIHBL : SelfLink
        from mShipIncomingHBLinks [1] to mShip.incomingHBLink [1];
```

```
}
behavior Admit;
behavior Touchup;
behavior MarkForRecycling;
behavior Ship;
```

The instantiation procedure in A.3.3 is amended for decisions and merges:

5. For each connector above
    a. .., b. ..., c. ..., ...

   For succession connectors with decision steps as their source or merge steps as their target
    - Create the same number of association atoms as there are values of all the connected features opposite the decision or merge step (1 for each in this example).
    - Assign each connected feature value as participant in exactly one assocation atom.
      If all the connected features together opposite the decision or merge step has more or fewer values than the decision or merge step, create atoms for the type of the feature with fewer values up to the number in the other feature, and assign them as values of the first feature.

The instantiation steps above produce the following atoms, choosing in this execution to touchup, rather than mark for recycling. Only the final result of execution is shown, at the end for brevity (MyManufacture), while the other atoms appear in the order created.

```
  /* Before decision. */
#atom
behavior MyAdmit specializes Admit;

  /* Decision. */
#atom
behavior MyInspect specializes DecisionPerformance;
#atom
assoc MyAdmit_Before_Inspect_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyAdmit;
    end feature redefines laterOccurrence : MyInspect;
}
  /* One decision branch taken. */
#atom
behavior MyTouchup specializes Touchup;
#atom
assoc MyInspect_Before_Touchup_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyInspect;
    end feature redefines laterOccurrence : MyTouchup;
}
  /* One merge branch taken. Merge. */
#atom
behavior MyMergeToShip specializes MergePerformance;
#atom
assoc MyTouchup_Before_Merge_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyTouchup;
    end feature redefines laterOccurrence : MyMergeToShip;
}
  /* After merge. */
#atom
behavior MyShip specializes Ship;
#atom
assoc MyMerge_Before_Ship_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyMergeToShip;
    end feature redefines laterOccurrence : Ship;
}
```

```
behavior MyManufactureSteps unions MyAdmit, MyInspect, MyTouchup, MyMergeToShip, MyShip;

#atom
behavior MyManufacture specializes Manufacture {
    feature redefines timeEnclosedOccurrences : MyManufactureSteps [5];

      /* Before decision. */
    step redefines admit : MyAdmit [1];

      /* Decision. */
    step redefines inspect : MyInspect [1];
    succession redefines a_before_i : MyAdmit_Before_Inspect_Link [1]
        first admit then inspect;

      /* One decision branch taken. */
    step redefines finish : MyTouchup [1];
    succession redefines i_before_f : MyInspect_Before_Touchup_Link [1]
        first inspect then finish;

      /* One merge branch taken. */
    succession redefines f_before_ms : MyTouchup_Before_Merge_Link [1]
        first finish then mShip;

      /* Merge. */
    step redefines mShip: MyMergeToShip [1];

       /* After merge */
    step redefines ship : MyShip [1];
    succession redefines ms_before_s : MyMerge_Before_Ship_Link [1]
      first mShip then ship;

      /* Decision and merge timing constraints. */
    feature redefines inspectOutgoingHBLinks : MyInspect_Before_Touchup_Link;
    feature redefines mShipIncomingHBLinks : MyTouchup_Before_Merge_Link;
}
```

A.3.8 Timing for behavior, Changing feature values

This covers changes in feature values of occurrences. Change execution requires creating additional atoms for the periods (*time slices*) when feature values are unchanged for each occurrence (*life*), and ordering the slices in time as feature values change. Time slice atoms are also occurrences, but represent the same thing as their life occurrence, just for a potentially smaller period of time, see 9.2.4.1.

Changes to occurrence feature values is modelled using the library behavior *FeatureWritePerformance* as a step, specifying when the change is to happen, see 9.2.8.1. Time slices are created each time it is performed.

The example below adds to the example in A.3.6.

- A class with changeable features (MyProduct with `isPainted`, `isDry`, `isShipped`).
- Features of behaviors to identify the above (`objectToFinish`).
- *FeatureWritePerformance* steps specifying when and how to change its feature values (in Paint, Dry, and Ship).

FeatureWritePerformances ensure when they finish that the occurrence has a time slice starting right then with the feature values specified, though the values might change immediately afterwards. Execution must define time slice atoms that prevent feature values from changing between *FeatureWritePerformances*, see below.

```
behavior Manufacture {
    feature objectToFinish : Product [1];
    step paint : Paint [1]{
      redefines objectToPaint = objectToFinish;
    }
    step dry : Dry [*]  {
      redefines objectToDry = objectToFinish;
    }
    succession p_before_d first paint [1] then dry [1];
    step ship : Ship [*]  {
      redefines objectToShip = objectToFinish;
    }
    succession d_before_s first dry [1] then ship [1];
}

struct Product {
    feature isPainted : Boolean [1] := false;
    feature isDry : Boolean [1] := true;
    feature isShipped : Boolean [1] := false;
}

behavior Paint {
    feature objectToPaint : Product [1];

    step painting : FeatureWritePerformance [1] {
      redefines onOccurrence : Product = objectToPaint {
        redefines startingAt : Product {
          redefines accessedFeature : Boolean [1] subsets isDry; } }
      redefines replacementValues = false;
    }

    succession p_before_p first painting [1] then painted [1];
    step painted : FeatureWritePerformance [*] {
      redefines onOccurrence : Product = objectToPaint {
        redefines startingAt : Product {
          redefines accessedFeature : Boolean [1] subsets isPainted; } }
      redefines replacementValues = true;
    }
}

behavior Dry {
    feature objectToDry : Product [1];
    step dried : FeatureWritePerformance [1] {
      redefines onOccurrence : Product = objectToDry {
        redefines startingAt : Product {
          redefines accessedFeature : Boolean [1] subsets isDry; } }
      redefines replacementValues = true;
    }
}

behavior Ship {
    feature objectToShip : Product [1];
    step shipped : FeatureWritePerformance [1] {
      redefines onOccurrence : Product = objectToShip {
        redefines startingAt : Product {
          redefines accessedFeature : Boolean [1] subsets isShipped; } }
      redefines replacementValues = true;
    }
}
```

The instantiation procedure is amended with

6. For behaviors with *FeatureWritePerformances*
    a. Create classes for time slices (ProductTimeSlice) specializing the kinds of things they are slicing (Product), and redefining the features being modified as `readonly` (`isPainted`, `isDry`, `isShipped`).
    b. Add time slice features to atoms of the kinds of things being modified (MyProduct):
        i. Before the first *FeatureWritePerformance* (`beforePaint`),
        ii. Between each successive *FeatureWritePerformance* (`whilePainting`, `afterPaint`, `afterDry`),
        iii. After the last *FeatureWritePerformance* (`afterShip`)
    c. In the behavior atom using *FeatureWritePerformance* (MyManufacture), create atoms for the above time slice features in order, assigning values to all the features, even if they were not modified, and specify that
        i. The first time slice above (i)
            - Starts (*`startShot`*) at the same time (*`timeCoincidentOccurrences`*) as the behavior.
            - Ends just before (*`immediateSuccessors`*) the first *FeatureWritePerformance* does (`paint.painting.endShot`).
        ii. The middle time slices (ii)
            - Start at the same time a *FeatureWritePerformance* ends.
            - End just before the next one does.
        iii. The last time slice (iii)
            - Starts at the same time the last *FeatureWritePerformance* ends (`ship.shipped.endShot`).
            - Ends at the same times as the behavior.

The instantiation steps above produce the following atoms, adding to (or modifying) those created in A.3.3. Step 6.a produces

```
#atom
struct MyProduct specializes Product;
#atom
behavior MyManufacture specializes Manufacture;

struct ProductTimeSlice specializes Product {
    readonly feature redefines isPainted;
    readonly feature redefines isDry;
    readonly feature redefines isShipped;
}
```

Instantiation step 6.b.i. the start of 6.c.i produces the first time slice (`beforePaint`) and starts it.

```
#atom
struct MyProduct specializes Product {
    feature beforePaint : ProductTimeSlice [1] subsets timeSlices;
}
#atom
behavior MyManufacture specializes Manufacture {
    feature redefines objectToFinish : MyProduct;
    feature startShot
        subsets objectToFinish.beforePaint.startShot.timeCoincidentOccurrences;
    feature obPiP chains objectToFinish.beforePaint.isPainted = false;
    feature obPiD chains objectToFinish.beforePaint.isDry = true;
    feature obPiS chains objectToFinish.beforePaint.isShipped = false;
}
```

The first of instantiation step 6.b.ii and first start of 6.c.ii produces the second time slice (`whilePainting`), while the end of 6.c.i ends the first (`beforePaint`). The MyManufacture features above are omitted for brevity.

```
#atom
struct MyProduct specializes Product {
    feature beforePaint : ProductTimeSlice [1] subsets timeSlices;
    feature whilePainting : ProductTimeSlice [1] subsets timeSlices;
}

behavior MyProductFeatureWrite specializes FeatureWritePerformance {
    feature redefines onOccurrence : MyProduct;
}
#atom
behavior PaintingMyProductFeatureWrite specializes MyProductFeatureWrite;
#atom
behavior MyPaint specializes Paint {
    feature redefines objectToPaint : MyProduct;
    step redefines painting : PaintingMyProductFeatureWrite;
}
#atom
behavior MyManufacture specializes Manufacture {
    ...
    step redefines paint : MyPaint;
    feature subsets objectToFinish.beforePaint.immediateSuccessors,
                    objectToFinish.whilePainting.startShot.timeCoincidentOccurrences
            chains paint.painting.endShot;
    feature owPiP chains objectToFinish.whilePainting.isPainted = false;
    feature owPiD chains objectToFinish.whilePainting.isDry = false;
    feature owPiS chains objectToFinish.whilePainting.isShipped = false;
}
```

The second of instantiation step 6.b.ii and second start of 6.c.ii produces the third time slice (`afterPaint`), while the end of the first 6.c.ii ends the second (`whilePainting`).

```
#atom
struct MyProduct specializes Product {
    feature beforePaint : ProductTimeSlice [1] subsets timeSlices;
    feature whilePainting : ProductTimeSlice [1] subsets timeSlices;
    feature afterPaint : ProductTimeSlice [1] subsets timeSlices;
}
#atom
behavior PaintedMyProductFeatureWrite specializes MyProductFeatureWrite;
#atom
assoc MyPaintingFW_Before_PaintFW_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : PaintingMyProductFeatureWrite;
    end feature redefines laterOccurrence : PaintedMyProductFeatureWrite;
}
#atom
behavior MyPaint specializes Paint {
    feature redefines objectToPaint : MyProduct;
    step redefines painting : PaintingMyProductFeatureWrite;
    step redefines painted : PaintedMyProductFeatureWrite;
    succession redefines p_before_p : MyPaintingFW_Before_PaintFW_Link
        first painting then painted;
}
#atom
behavior MyManufacture specializes Manufacture {
    ...
    feature subsets objectToFinish.whilePainting.immediateSuccessors,
```

```
                    objectToFinish.afterPaint.startShot.timeCoincidentOccurrences
            chains paint.painted.endShot;
    feature oaPiP chains objectToFinish.afterPaint.isPainted = true;
    feature oaPiD chains objectToFinish.afterPaint.isDry = false;
    feature oaPiS chains objectToFinish.afterPaint.isShipped = false;
}
```

The third of instantiation step 6.b.ii and third start of 6.c.ii produces the fourth time slice (`afterDry`), while the end of the second 6.c.ii ends the third (`afterPaint`).

```
#atom
struct MyProduct specializes Product {
    feature beforePaint : ProductTimeSlice [1] subsets timeSlices;
    feature whilePainting : ProductTimeSlice [1] subsets timeSlices;
    feature afterPaint : ProductTimeSlice [1] subsets timeSlices;
    feature afterDry : ProductTimeSlice [1] subsets timeSlices;
}
#atom
behavior MyDry specializes Dry {
    feature redefines objectToDry : MyProduct;
    step redefines dried : MyProductFeatureWrite;
}
#atom
assoc MyPaint_Before_Dry_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyPaint;
    end feature redefines laterOccurrence : MyDry;
}
behavior MyManufacture specializes Manufacture {
    ...
    step redefines dry : MyDry;
    succession redefines p_before_d : MyPaint_Before_Dry_Link [1] first paint then dry;
    feature dry.dried.endShot
        subsets objectToFinish.afterPaint.immediateSuccessors,
                objectToFinish.afterDry.startShot.timeCoincidentOccurrences;
    feature oaDiP chains objectToFinish.afterDry.isPainted = true;
    feature oaDiD chains objectToFinish.afterDry.isDry = true;
    feature oaDiS chains objectToFinish.afterDry.isShipped = false;
}
```

Instantiation step 6.b.iii and 6.c.iii produce the fifth time slice (`afterShip`), while the end of the third 6.c.ii ends the fourth (`afterDry`).

```
#atom
struct MyProduct specializes Product {
    feature beforePaint : ProductTimeSlice [1] subsets timeSlices;
    feature whilePainting : ProductTimeSlice [1] subsets timeSlices;
    feature afterPaint : ProductTimeSlice [1] subsets timeSlices;
    feature afterDry : ProductTimeSlice [1] subsets timeSlices;
    feature afterShip : ProductTimeSlice [1] subsets timeSlices;
}
#atom
behavior MyShip specializes Ship {
    feature redefines objectToShip : MyProduct;
    step redefines shipped : MyProductFeatureWrite;
}
#atom
assoc MyDry_Before_Ship_Link specializes HappensBefore {
    end feature redefines earlierOccurrence : MyDry;
    end feature redefines laterOccurrence : MyShip;
```

```
}
#atom
behavior MyManufacture specializes Manufacture {
    ...
    step redefines ship : MyShip;
    succession redefines d_before_s : MyDry_Before_Ship_Link [1] first dry then ship;
    feature subsets objectToFinish.afterDry.immediateSuccessors,
                    objectToFinish.afterShip.startShot.timeCoincidentOccurrences
            chains ship.shipped.endShot;
    feature redefines endShot subsets objectToFinish.afterShip.timeCoincidentOccurrences;
    feature oaSiP chains objectToFinish.afterShip.isPainted = true;
    feature oaSiD chains objectToFinish.afterShip.isDry = true;
    feature oaSiS chains objectToFinish.afterShip.isShipped = true;
}
```