Date: June 2015

OMG Meta Object Facility (MOF) Core Specification

Version 2.5

OMG Document Number: formal/2015-06-05
Standard document URL: http://www.omg.org/spec/MOF/2.5
Associated Normative Machine-Readable Files:

http://www.omg.org/spec/MOF/20131001/MOF.xmi

Associated Non-normative Machine-Readable Files:

http://www.omg.org/spec/MOF/20131001/CMOFConstraints.ocl
http://www.omg.org/spec/MOF/20131001/EMOFConstraints.ocl

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

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 *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/report_issue.htm).

Table of Contents

1 Scope 1
2 Conformance 1
3 Normative References 1
4 Terms and Definitions 2
5 Symbols 2
6 Additional Information 2
6.1 General Information 2
6.2 Structure of the MOF 2 Specification 3
7 MOF Architecture (informative) 5
7.1 General 5
7.2 MOF 2 Design Goals 5
7.3 How Many Meta Layers? 6
7.4 Reuse of the Common Metamodel for UML and MOF 7
8 Language Formalism 9
8.1 General 9
8.2 Metamodel Specification 9
8.3 Using Packages to Partition and Extend Metamodels 9
9 Reflection 11
9.1 General 11
9.2 Element 11
9.3 Factory 13
9.4 Object 14
10 Identifiers 17
10.1 General 17
10.2 Extent 17
10.3 URIExtent 18
10.4 MOF::Common 19
10.5 ReflectiveCollection 19
10.6 ReflectiveSequence 20
11 Extension 23
11.1 General 23

11.2 Tag 23
12 The Essential MOF (EMOF) Model 25
12.1 General 25
12.2 EMOF Merged Model 26
12.3 Merged Elements from MOF 28
12.4 EMOF Constraints 29
12.5 EMOF Definitions and Usage Guidelines for the UML Models 31
12.6 Predefined Tags 32
13 CMOF Reflection 35
13.1 General 35
13.2 Link 36
13.3 Argument 37
13.4 Object 37
13.5 Element 38
13.6 Factory 38
13.7 Extent 39
14 The Complete MOF (CMOF) Model 41
14.1 General 41
14.2 Elements used from UML 2 41
14.3 Imported Elements from MOF 43
14.4 CMOF Constraints 43
14.5 CMOF Extensions to Capabilities 45
15 CMOF Abstract Semantics 47
15.1 General 47
15.2 Approach 47
15.3 MOF Instances Model 47
15.4 Remarks on MOF Instance Modeling 50
15.5 Object Capabilities 50
15.6 Link Capabilities 51
15.7 Factory Capabilities 52
15.8 Extent Capabilities 53
15.9 Additional Operations 55
Annex A - XMI for MOF 2 Core 59
Annex B - Metamodel Constraints in OCL 61
Annex C - Migration from MOF 1.4 63
Annex D - Bibliography 69

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Formal Specifications are available from this URL:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- CORBA/IIOP
- Data Distribution Services
- Specialized CORBA

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- UML, MOF, CWM, XMI
- UML Profile

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- CORBAServices
- CORBAFacilities

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

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

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

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier - 10 pt. Bold: Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/report_issue.htm*.

1 Scope

This International Standard provides the basis for metamodel definition in OMG's family of MDA languages and is based on a simplification of UML2's class modeling capabilities. In addition to providing the means for metamodel definition it adds core capabilities for model management in general, including Identifiers, a simple generic Tag capability and Reflective operations that are defined generically and can be applied regardless of metamodel.

MOF 2 Core is built on by other OMG MOF specifications, including the following (in this list 'MOF based model' means any model that instantiates a metamodel defined using MOF, which includes metamodels themselves):

- XMI - for interchanging MOF-based models in XML [XMI25]
- MOF 2 Facility and Object Lifecycle - for connecting to and managing collections of MOF-based model elements [MOFFOL]
- MOF 2 Versioning and Development Lifeycle - for managing versions and configurations of MOF-based models [MOFVD]
- MOF Queries Views and Transformations - for transforming MOF-based models [QVT]
- MOF Models to Text - for generating text, such as programs, from MOF-based models [MOFM2T]
- Object Constraint Language - for specifying constraints on MOF-based models [OCL]

2 Conformance

There are two compliance points:

- Essential MOF (EMOF)
- Complete MOF (CMOF)

Compliant implementations may support EMOF only, see 12.4 for further detail, or may support CMOF, which includes EMOF. See 14.4 for detail.

All compliant implementations shall conform to the MOF Platform-Independent Model specified in Clause 15 and support the technology mapping specified in the XML Metadata Interchange (XMI) specification [XMI25].

3 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Readers of this MOF 2 Core specification are expected to be familiar with the UML specification since UML provides the structures forming MOF metamodels. Since version 2.4, UML and MOF share a common metamodel, the UML metamodel. The lightweight subset representing the MOF meatmodel is specified by specifying constraints against the UML metamodel. Diferent sets of constraints are used to construct the EMOF and CMOF compliance levels.

Hence the normative reference is the Classes clause of the following specification:

- [UML25] Unified Modeling Language (OMG UML), http://www.omg.org/spec/UML/2.5/

The mandatory mapping of MOF to XMI is specified at:

- [XMI25] XML Metadata Interchange - http://www.omg.org/spec/XMI/2.5

Formal constraints are expressed in OCL, which is specified at:

- [OCL] ISO/IEC 19507:2012 “Information technology - Object Management Group Object Constraint Language (OCL)” (OMG Object Constraint Language (OCL) - http://www.omg.org/spec/OCL/2.3.1)

The following reference is used in MOF1 to MOF2 migration Annex:

- [MOF1] ISO/IEC 19502:2005 “Meta Object Facility (MOF) Specification Version 1.4.1” (OMG Meta Object Facility (MOF) Specification (Version 1.4) - http://www.omg.org/spec/MOF/1.4)

4 Terms and Definitions

NULL

- Null is used in this International Standard to indicate the absence of a value. for example, a single-valued property that is null has no value, and when an operations returns null, it is returning no value.

5 Symbols

MOF 2 reuses the subset of the structural modeling symbols from UML 2 that is needed for class modeling. MOF 2 does not define any additional symbols.

See the UML specification [UML25] for the symbol definitions.

6 Additional Information

6.1 General Information

Incompatible and often proprietary metadata across different systems is a primary limitation on data exchange and application integration. Metadata are data about data. They are the data used by tools, databases, middleware, etc. to describe structure and meaning of data.

The Meta Object Facility (MOF) provides an open and platform-independent metadata management framework and associated set of metadata services to enable the development and interoperability of model and metadata driven systems. Examples of systems that use MOF include modeling and development tools, data warehouse systems, metadata repositories, etc.

MOF has contributed significantly to the core principles of the OMG Model Driven Architecture. Building on the modeling foundation established by UML, MOF introduced the concept of formal metamodels and Platform Independent Models (PIM) of metadata (examples include several st andard OMG metamodels including UML, MOF itself, CWM, SPEM, Java EJB, EDOC, EAI, etc.) as well as mappings from PIMs to specific platforms (Platform Specific Models and mapping examples

include MOF-to-Text mapping in the MOF-to-Text specification [MOFM2T], MOF-to-XML mapping in the XMI specification [XMI24], MOF-to-XML Schema mapping in the XMI production of XML Schema specification [XMI24], and MOF-to-Java in the JMI specification).

The OMG adopted the MOF 1.1 specification in November 1997 coincident with the adoption of UML 1.1. In March 2003, the significantly re-architected MOF 2.0 Core was adopted by the OMG, aligned with the then also adopted UML 2.0. The alignment between MOF and UML was then completed with MOF 2.4 and UML 2.4 by sharing the same metamodel for the definition of UML and MOF, using OCL constraints to define the metamodel subset relevant for MOF. The resulting MOF specification is presented in this document and referred to as "MOF 2" to distinguish it from the MOF 1.4 specification, which continues to exist as OMG specification and the ISO/IEC 19505 international standard.

MOF 2 is represented by a set of specifications: MOF 2 Core [MOF2], MOF 2 XMI Mapping (now titled XML Metadata Interchange) [XMI24], MOF 2 Facility and Object Lifecycle [MOFFOL], MOF 2 Versioning and Development Lifecycle [MOFVD], MOF 2 Query/View/Transformations [QVT], MOF Model to Text [MOFM2T].

6.2 Structure of the MOF 2 Specification

MOF 2 reuses the structural modeling capabilities of UML 2, based on the common metamodel shared between UML 2 and MOF 2. The OCL constraints limiting this metamodel to the MOF 2 - relevant subsets are defined in Clause for EMOF and Clause for CMOF. A reference to files with the OCL source code is provided in Annex.

Clause 7 provides an introduction to metamodeling and the MOF 2 architecture. Clause 8 introduces MOF 2 as a metamodeling language.

MOF 2 Core extends the shared metamodel with MOF 2 - specific capabilities. These are defined in Clauses 9 to 11 of this document. These MOF 2 capabilities are:

- Reflection: Extends a model with the ability to be self-describing.
- Identifiers: Provides an extension for uniquely identifying metamodel objects without relying on model data that may be subject to change.
- Extension: a simple means for extending model elements with name/value pairs. The following clauses describe each of the packages making up the supported capabilities.

The various packages making up the MOF 2 capabilities are instances of CMOF::Package, and all of its contents are instances of classes in the CMOF Model.

A metamodel is a model used to model modeling itself. The MOF 2 Model is used to model itself as well as other models and other metamodels (such as UML 2 and CWM 2, etc.). A metamodel is also used to model arbitrary metadata (for example, software configuration or requirements metadata).

The MOF 2 Model is made up of two main packages, Essential MOF (EMOF) and Complete MOF (CMOF). EMOF is described in Clause 12. EMOF is designed to match the capabilities of object oriented programming languages and of mappings to XMI or JMI. The Complete MOF (CMOF) provides the full metamodeling capabilities of MOF 2. To enable this, CMOF extends Reflection as described in Clause 13. CMOF itself is then defined in Clause 14.

Clause 15 provides an abstract instance model of MOF 2, effectively providing a Platform-Independent Model for CMOF.

Annex A provides links to machine-consumable definitions of EMOF and CMOF using XMI.

MOF 2 shares its metamodel with UML 2. Annex B provides links to the OCL source files to constrain the UML 2 metamodel down to EMOF or CMOF.

Annex C provides the normative mapping for the migration from MOF 1.4 to MOF 2.

Annex D contains the bibliographic references for the citations throughout this document. The format used for these citations is “[xyz].”

7 MOF Architecture (informative)

7.1 General

This clause describes the architecture of the MOF and how it serves as the platform-independent metadata management foundation for MDA. It also summarizes major architectural decisions that influenced the design of MOF 2. Finally, the relationship of MOF 2 to UML 2 and the use of MOF to instantiate UML 2 and future OMG metamodels is summarized. MOF 2 is a new generation of MOF [MOF2], aligned with UML 2 [UML25]; it does not replace MOF 1.4 [MOF1] which is aligned with UML 1.4 [UML1].

7.2 MOF 2 Design Goals

The primary purpose of this major revision of MOF is to provide a next-generation platform-independent metadata framework for OMG that builds on the unification accomplished in MOF 1.4, XMI 1.2, XMI production of XML Schemas, and JMI 1.0. The modeling foundation of the MOF 2 has strongly influenced and, at the same time, has been strongly influenced by the UML 2 Infrastructure specification because of a shared vision of reusing the core modeling concepts between UML 2, MOF 2, and other emerging OMG metamodels. The fact that some of the same companies and designers worked on both the specifications and championed these design principles has made this unification and reuse possible. Since version 2.4 the unification between UML and MOF has been further improved by basing MOF on the UML metamodel. Continuing the tradition of MOF since 1997, MOF2 can be used to define and integrate a family of metamodels using simple class modeling concepts. As in MOF1 only UML class modeling notation is used to describe MOF compliant metamodels. What is significant about MOF2 is that we have unified the modeling concepts in MOF2 and UML2 and reused a common metamodel in both the MOF2 and UML2 specifications. The major benefits of this approach include:

- Simpler rules for modeling metadata (just understand a subset of UML class modeling without any additional notations or modeling constructs).
- Various technology mappings from MOF (such as XMI, JMI, etc.) now also apply to a broader range of UML models including UML profiles.
- Broader tool support for metamodeling (any UML modeling tool can be used to model metadata more easily).

In any case MOF2 can be used to define (without the need to reuse specific metamodel packages) both object and non-object oriented metamodels (as was true with MOF1).

Based on the experience of implementers of MOF, XMI, and JMI in the context of well known industry standard metamodels such as UML and CWM, some of the overriding design concerns and goals are:

1. Ease of use in defining and extending existing and new metamodels and models of software infrastructure. We wanted to make sure that defining and extending metamodels and models of metadata is as simple as defining and extending normal object models. The reuse of a ‘common core’ between UML 2, MOF 2, and additional key meta-models (CWM, EAI) is key to accomplishing this goal. Future RFPs for CWM2, EAI2, etc. are expected to either reuse the common core or propose changes to improve the reusability.
2. Making the MOF model itself much more modular and reusable. In a sense, we have begun the work of component-oriented modeling where model packages themselves become reusable across modeling frameworks. The roots of this work began when CWM was being defined. The complexity of modeling the data warehousing problem domain necessitated this divide-and-conquer approach. The refactoring done so far has resulted in a more modular set of packages suitable for object modeling.

3. The use of model refactoring to improve the reusability of models. Some of the lessons learned were influenced by the refactoring experience from the programming language domain at the class level that is much more widely used. Also influencing our work was the experience of the CWM design team and the UML 2 design teams. While this approach has resulted in a larger number of fine grained packages, we believe this approach will improve reuse and speed the development of metamodels and new modeling frameworks. A direct result of this effort is the reuse of a subset of 'Common Core' metamodel packages by MOF 2 and UML 2 Specifications.

4. Ensure that MOF 2 is technology platform independent and that it is more practical to map from MOF 2 to a number of technology platforms such as J2EE, .Net, CORBA, Web Services, etc. The experience gained in the definition of the MOF, XMI, and JMI specifications, which already define many technology mappings from and to the MOF model, has been a solid foundation for this effort. It is a design goal that MOF implementations using different language mappings can interoperate (for example, using XML interchange).

5. Orthogonality (or separation of concerns) of models and the services (utilities) applied to models is a very important goal for MOF 2. One of the lessons learned in MOF 1 and XMI 1 was that the original design of MOF was overly influenced and constrained by the assumed lifecycle semantics of CORBA based metadata repositories. As it turned out, the industry embraced a more loosely coupled way to interchange metadata (as well as data) as evidenced by the popularity of XML and XMI. Interestingly, vendors used MOF in many different ways - to integrate development tools, data warehouse tools, application management tools, centralized and distributed repositories, developer portals, etc. It became clear that to provide implementation flexibility, we had to decouple the modeling concepts from the desirable metadata services such as metadata interchange (using XML streams versus using Java/CORBA objects), Reflection, Federation, Life Cycle, Versioning, Identity, Queries, etc. We consider this orthogonality of models from services to be a very significant feature of MOF 2. Because of the variety of implementation choices and services available, many of the more complex services (Federation, Versioning, Query, etc.) are subjects of additional OMG RFPs.

6. MOF 2 models reflection using MOF itself as opposed to just specifying reflection as a set of technology-specific interfaces. This is in the spirit of item 5 above to model Reflection as an independent service. This approach also clearly separates the concerns of reflection, life cycle management, etc., which were combined together in MOF 1.

7. MOF 2 models the concept of identifier. The lack of this capability in MOF, UML, CWM, etc., made interoperability of metadata difficult to implement. The authors understand that modeling identifiers is not easy, but plan to show its usefulness in a simple domain - identifiers for metadata first. A key design goal is to make it easy to map this model of identifier to W3C identifier and referencing mechanisms such as the URI.

8. Reuse of modeling frameworks and model packages at various metalayers by better packaging of MOF 'Capabilities.' Note that some commonly used types and services can be used in defining the MOF itself, various metamodels (such as UML and CWM), as well as user models and even user objects. A by-product of the orthogonality principle is that some MOF capabilities can be used at multiple metalayers.

7.3 How Many Meta Layers?

One of the sources of confusion in the OMG suite of standards is the perceived rigidness of a 'Four layered metamodel architecture' that is referred to in various OMG specifications. Note that key modeling concepts are Classifier and Instance or Class and Object, and the ability to navigate from an instance to its metaobject (its classifier). This fundamental concept can be used to handle any number of layers (sometimes referred to as metalevels). The MOF 2 Reflection interfaces allow traversal across any number of metalayers recursively. Note that most systems use a small MOF Core Specification, v2.0 9 number of levels (usually less than or equal to four). Example numbers of layers include 2 (generic reflective systems - Class/Object), 3 (relational database systems - SysTable/Table/Row), and 4 (UML 2 Infrastructure, UML 1.4, and MOF 1.4 specification - MOF/UML/User Model/User Object). MOF 1 and MOF 2 allow

any number of layers greater than or equal to 2. (The minimum number of layers is two so we can represent and navigate from a class to its instance and vice versa). Suffice it to say MOF 2 with its reflection model can be used with as few as 2 levels and as many levels as users define.

7.4 Reuse of the Common Metamodel for UML and MOF

Since version 2.4 for UML and MOF Core, MOF reuses the UML metamodel, narrowed down to the specific subsets for EMOF and CMOF by applying constraints. See corresponding elements of the UML metamodel by using PackageMerge. This leaves the hierarchy of metaclasses in the shared UML metamodel undisturbed, since PackageMerge is not subclassing. Instead, features of the corresponding elements residing in the to be merged packages are simply combined into one single element conveyed into the resulting package.

Standard class modeling concepts (importing, subclassing, adding new classes, associations, and adding associations between existing classes) are used for MOF 2 extensibility. (This is identical to the extensibility mechanism in MOF 1). These concepts are used to define additional packages (such as Reflection, Extents, and Identities) in MOF 2 as well as in any other MOF 2 compliant model.

Figure 7-1 shows how MOF reuses and extends the UML Core metamodel.

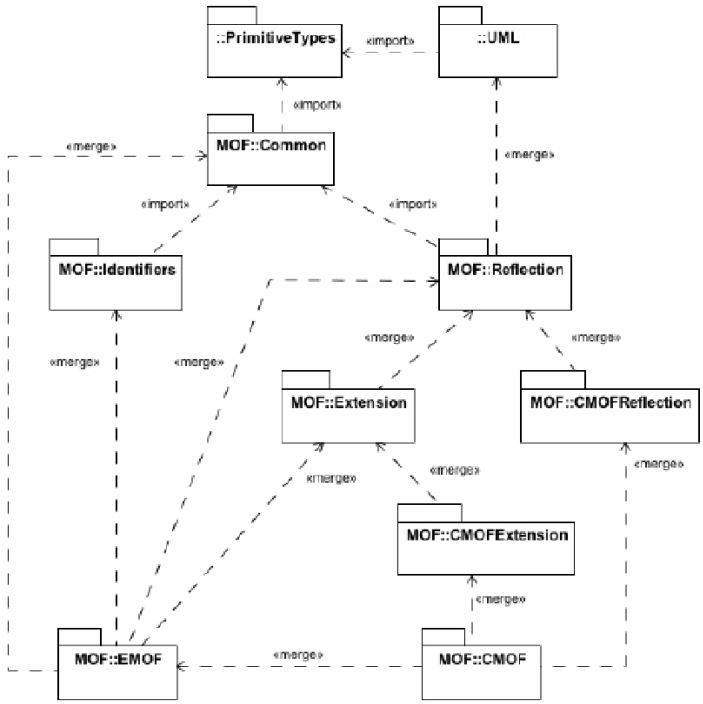

Figure 7-1 - MOF imports from the UML Core

8 Language Formalism

8.1 General

This clause explains techniques used to describe the MOF. The MOF is described using both textual and graphic presentations. The International Standard uses a combination of languages (a subset of UML, an object constraint language, and precise natural language) to precisely describe the abstract syntax and semantics of the MOF. Unlike MOF 1 and UML 1 where slightly different techniques (sometimes subtly different) were used, the MOF 2 specification reuses much of the formalisms in the UML 2 specification. In particular, EMOF and CMOF are both described using CMOF, which is also used to describe UML2. EMOF is also completely described in EMOF by applying package import, and merge semantics from its CMOF description. As a result, EMOF and CMOF are described using themselves, and each is derived from the UML 2 metamodel.

8.2 Metamodel Specification

Please refer to the UML specification [UML25] regarding the language formalism used for UML and MOF. The CMOF model follows that formalism closely by merging the UML packages and applying constraints to select the CMOF subset within the UML metamodel. The formalism for EMOF is slightly simpler by selecting a narrower subset from the UML metamodel.

8.3 Using Packages to Partition and Extend Metamodels

Packages can be used for two purposes. The first is package import; a mechanism for grouping related model elements together in order to manage complexity and facilitate reuse. Since a Package is also a Namespace, model elements referenced across package boundaries must be qualified by their full package name. Package import relaxes this constraint by making model elements in the imported package directly visible in the importing package where they may be used in associations with other model elements, specialized with sub-classes that provide additional features, etc. The second use of packages is to facilitate combining new or reusable metamodeling features to create extended modeling languages. Package merging combines the features of the merged package with the merging package to define new integrated language capabilities. After package merge, classes in the merging package contain all the features of similarly named classes in the merged package.

9 Reflection

9.1 General

Metaobjects enable use of objects without prior knowledge of the objects' specific features. In a MOF context, an object's class (i.e., its metaobject) reveals the nature of the object - its kind, its features. The Reflection Package allows this discovery and manipulation of metaobjects and metadata.

Figure 9.1 shows the MOF Reflection architecture. The classes on the left side of the diagram are introduced or extended by MOF::Reflection, while the right side of the diagram shows the classes shared with the UML metamodel. That relationship is also illustrated in the embedded Package diagram.

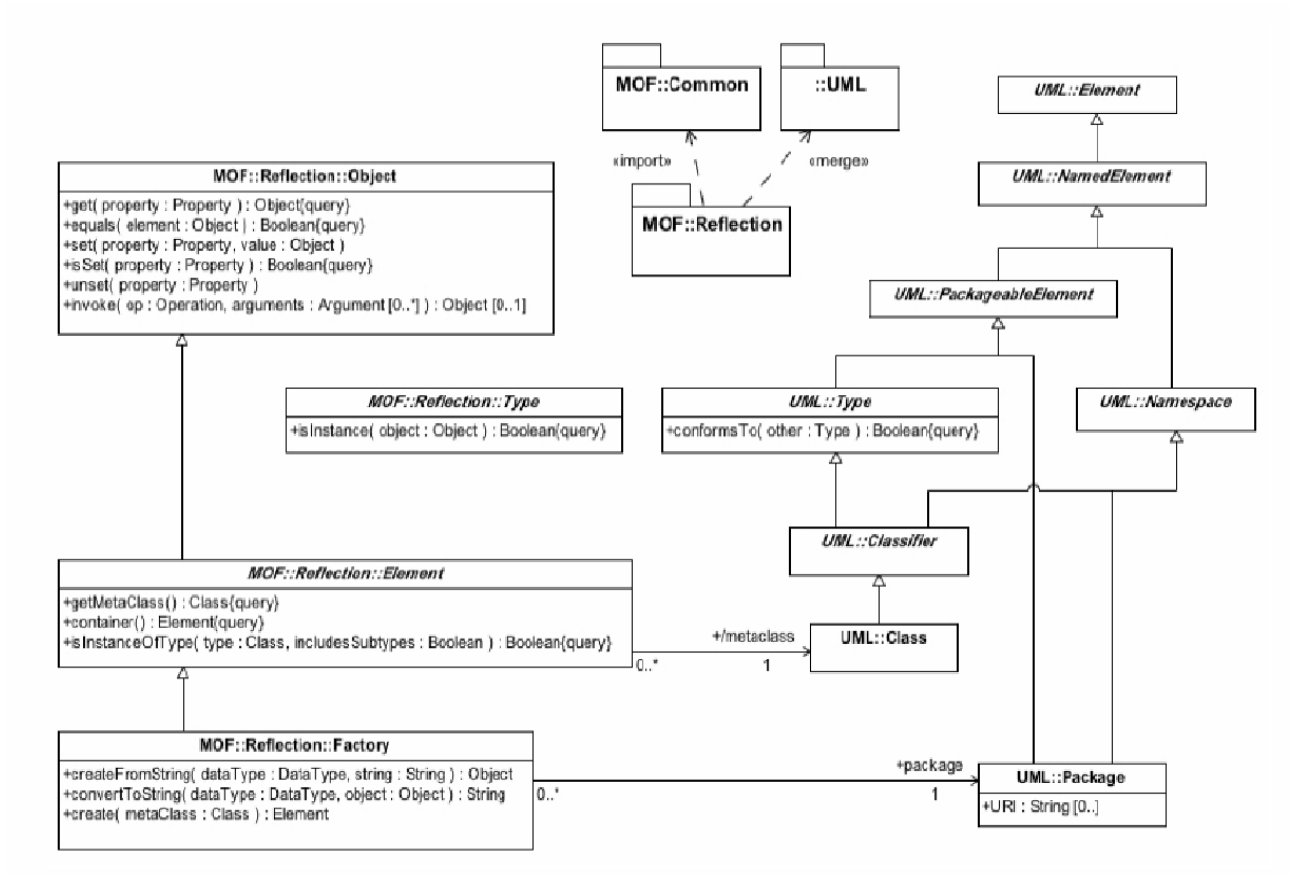

Figure 9.1- The Reflection Package

9.2 Element

Every Element has a Class that describes its properties and operations. The Element is an Instance of this Class. Element merges and extends UML::Element. All model elements that specialize Reflection::Element inherit reflective capabilities. In particular, this includes all model elements from UML.

All the conditions in 12.5 are also part of the reflective behavior.

If any reflective operation attempts to create cyclic containment, an IllegalArgumentException is thrown.

Properties

/metaclass: Class

Returns the Class that describes this element. This is a derived property provided for convenience and consistency.

Operations

getMetaClass() : Class

Returns the Class that describes this element.

container(): Element

Returns the parent container of this element if any. Return Null if there is no containing element.

Constraints

No additional constraints.

Semantics

Class Element is the superclass of all classes defined in MOF, and is an implicit superclass of all metaclasses defined using MOF: this superclass relationship to Element does not need to be explicitly modeled in MOF-compliant metamodels, and if implicit in this way Element is not included in the list of superclasses.

By creating Properties with type Element it is possible to reference elements in any MOF-compliant model, similar to the use of xsd:any in XML Schemas.

Each element can access its metaClass in order to obtain a Class that provides a reflective description of that element. By having both MOF and instances of MOF be rooted in class Element, MOF supports any number of meta layers as described in Clause 7.

The following describes the interaction between default values, null, isSet, and unSet.

Single-valued properties

If a single-valued property has a default:

- It is set to that default value when the element is created. isSet=false.
- If the value of that property is later explicitly set, isSet=true, unless it is set to the default value (if any) in which case isSet=false.
- If the property is unSet, then the value of the property returns to the default, and isSet=false.

If a single-valued property does not have a default:

- At creation, its value is null. isSet=false.
- If the value of that property is later explicitly set, even to null, isSet=true.
- If the property is unSet, then the value of the property returns to null, and isSet=false.

Multi-valued properties:

- When the element is created, it is an empty list (isSet=false).
- If the list is modified in any way (except unSet), isSet=true.
- If the list is unSet, it is cleared and becomes an empty list (isSet=false).

The implementation of isSet is up to the implementer. For default values, implementations are not required to access stored metadata at runtime. It is adequate to generate a constant in the implementation class for the default.

Rationale

Element is introduced in package Reflection so that it can be combined with Core::Basic to produce EMOF, which can then be merged into CMOF to provide reflective capability to MOF and all instances of MOF.

9.3 Factory

An Element may be created from a Factory. A Factory is an instance of the MOF Factory class. A Factory creates instances of the types in a Package.

Properties

- package: Package [1] Returns the package this is a factory for.

Operations

createFromString(dataType: DataType, string: String): Object

Creates an Object initialized from the value of the String. Returns null if the creation cannot be performed. The format of the String is defined by the XML Schema SimpleType corresponding to that datatype.

- Exception: NullPointerException if datatype is null.
- Exception: IllegalArgumentException if datatype is not a member of the package returned by getPackage().

convertToString(datatype: DataType, object: Object): String

Creates a String representation of the object. Returns null if the creation cannot be performed. The format of the String is defined by the XML Schema SimpleType corresponding to that dataType.

- Exception: IllegalArgumentException if datatype is not a member of the package returned by getPackage() or the supplied object is not a valid instance of that datatype.

create(metaClass: Class): Element

Creates an element that is an instance of the metaClass. Object::metaClass == metaClass and metaClass.isInstance(object) == true.

All properties of the element are considered unset. The values are the same as if object.unset(property) was invoked for every property.

Returns null if the creation cannot be performed. Classes with abstract = true always return null.

The created element's metaClass == metaClass.

- Exception: NullPointerException if class is null.
- Exception: IllegalArgumentException if class is not a member of the package returned by getPackage().

Constraints

The following conditions on metaClass: Class and all its Properties must be satisfied before the metaClass: Class can be instantiated. If these requirements are not met, create() throws exceptions as described above.

[1] Meta object must be set.

[2] Name must be 1 or more characters.

[3] Property type must be set.

[4] Property: 0 <= LowerBound <= UpperBound required.

[5] Property: 1 <= UpperBound required.

[6] Enforcement of read-only properties is optional in EMOF.

[7] Properties of type Class cannot have defaults.

[8] Multivalued properties cannot have defaults.

[9] Property: Container end must not have upperBound >1, a property can only be contained in one container.

[10] Property: Only one end may be composite.

[11] Property: Bidirectional opposite ends must reference each other.

[12] Property and DataType: Default value must match type.

Items 3-12 apply to all Properties of the Class.

These conditions also apply to all superclasses of the class being instantiated.

9.4 Object

Reflection introduces Object as a supertype of Element in order to be able to have a Type that represents both elements and data values. Object represents ‘any’ value and is the equivalent of java.lang.Object in Java.

9.4.1 Operations

equals(object: Object): Boolean

Determines if the object equals this Object instance. For instances of Class, returns true if the object and this Object instance are references to the same Object. For instances of DataType, returns true if the object has the same value as this Object instance. Returns false for all other cases.

get(property: Property) : Object

Gets the value of the given property. If the Property has multiplicity upper bound of 1, get() returns the value of the Property. If Property has multiplicity upper bound >1, get() returns a ReflectiveCollection containing the values of the Property. If there are no values, the ReflectiveCollection returned is empty.

- Exception: throws IllegalArgumentException if Property is not a member of the Class from class().

set(property: Property, object: Object)

If the Property has multiplicity upper bound = 1, set() atomically updates the value of the Property to the object parameter. If Property has multiplicity upper bound >1, the Object must be a kind of ReflectiveCollection. The behavior is identical to the following operations performed atomically:

```
ReflectiveSequence list = element.get(property);
list.clear();
list.addAll((ReflectiveSequence) object);
```

There is no return value.

- Exception: throws IllegalArgumentException if Property is not a member of the Class from getMetaClass().
- Exception: throws ClassCastException if the Property's type isInstance(object) returns false and Property has multiplicity upper bound = 1.
- Exception: throws ClassCastException if Element is not a ReflectiveCollection and Property has multiplicity upper bound > 1.
- Exception: throws IllegalArgumentException if element is null, Property is of type Class, and the multiplicity upper bound > 1.

isSet(property: Property): Boolean

If the Property has multiplicity upper bound of 1, isSet() returns true if the value of the Property is different than the default value of that property. If Property has multiplicity upper bound >1, isSet() returns true if the number of objects in the list is > 0.

- Exception: throws IllegalArgumentException if Property is not a member of the Class from getMetaClass ().

unset(property: Property)

If the Property has multiplicity upper bound of 1, unset() atomically sets the value of the Property to its default value for DataType type properties and null for Class type properties. If Property has multiplicity upper bound >1, unset() clears the ReflectiveCollection of values of the Property. The behavior is identical to the following operations performed atomically:

```
ReflectiveCollection list = object.get(property);
list.clear();
```

There is no return value.

After unset() is called, object.isSet(property) == false.

- Exception: throws IllegalArgumentException if Property is not a member of the Class from getMetaClass().

10 Identifiers

10.1 General

An element has an identifier in the context of an extent that distinguishes it unambiguously from other elements.

There are practical uses for object identifiers. Identifiers can simplify serializing references to external objects for interchange. They can serve to coordinate data updates where there has been replication, and can provide clear identification of objects in communication, such as from user interfaces. Identifiers support comparing for identity where implementations might have multiple implementation objects that are to be considered, for some purposes, to be the same object. Identifiers also facilitate Model Driven Development by providing an immutable identifier that can be used to correlate model elements across model transformations where both the source and target models may be subject to change. Model to model reconciliation requires some means of determining how model elements were mapped that does not rely on user data (such as names) that may be subject to change.

Figure 10-1 shows the MOF Identifiers architecture. The classes on the left side of the diagram are introduced by MOF::Identifiers and derived from MOF::Object. They are MOF capabilities, not model elements. The right side of the diagram shows Package and Property, their URI and isID attributes originated in MOF, but are from this version on shared with the UML metamodel. The package diagram insert on the far right illustrates the position of the Identifiers package in the package stack.

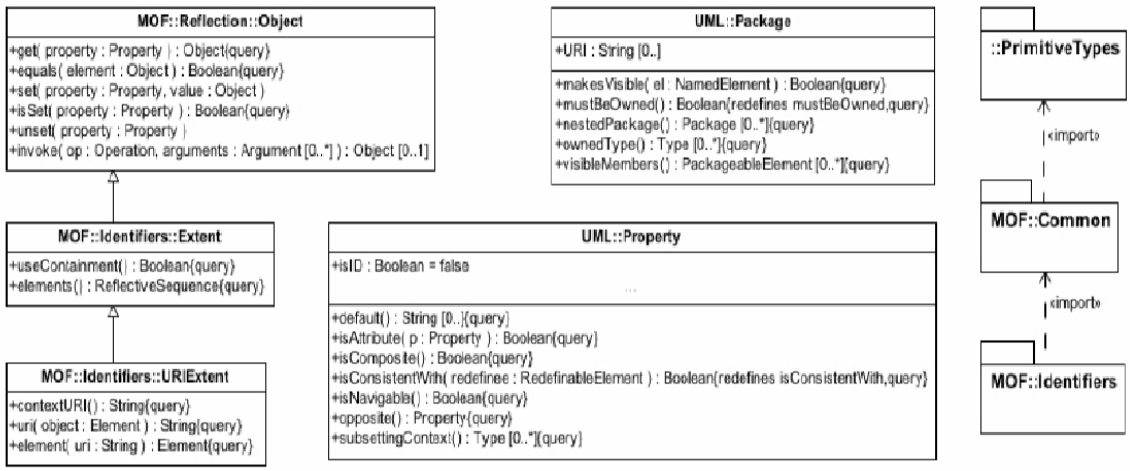

Figure 10.1 - The Identifiers package

10.2 Extent

An Extent is a context in which an Element in a set of Elements in a set can be identified. An element may be a member of zero or more extents. An Extent is not an Element, it is part of a MOF capability.

Properties

No additional properties.

Operations

useContainment(): Boolean

When true, recursively include all elements contained by members of the elements().

elements(): ReflectiveSequence

Returns a ReflectiveSequence of the elements directly referenced by this extent. If exclusive()==true, these elements must have container()==null. Extent.elements() is a reflective operation, not a reference between Extent and Element.

Constraints

No additional constraints.

Semantics

When the element is created, it is not assigned to any Extent.

Rationale

Extents provide a context in which MOF Elements can be identified independent of any value in the Element.

10.3 URIExtent

An extent that provides URI identifiers. A URIExtent can have a URI that establishes a context that may be used in determining identifiers for elements identified in the extent. Implementations may also use values of properties with isID==true in determining the identifier of the element.

Properties

No additional properties.

Operations

contextURI(): String

Specifies an identifier for the extent that establishes a URI context for identifying elements in the extent. An extent has an identifier if a URI is assigned. URI is defined in IETF RFC-2396 available at http://www.ietf.org/rfc/rfc2396.txt.

uri(element: Element): String

Returns the URI of the given element in the extent. Returns Null if the element is not in the extent.

element(uri: String): Element

Returns the Element identified by the given URI in the extent. Returns Null if there is no element in the extent with the given URI. Note the Element does not (necessarily) contain a property corresponding to the URI. The URI identifies the element in the context of the extent. The same element may have a different identifier in another extent.

Constraints

No additional constraints.

Semantics

The URI may incorporate the value of Properties that are marked as an identifier (isID==true).

Rationale

URIs are defacto standard identifiers. They are useful for identifying MOF elements and navigating links between them.

10.4 MOF::Common

Package MOF::Common contains MOF-internal features to handle multi-valued entities. These features, as shown in Figure 10.2, are used throughout MOF, but are not model elements. The package diagram inserted in Figure 10.2 illustrates the usage of the PrimitiveTypes package shared between UML and MOF.

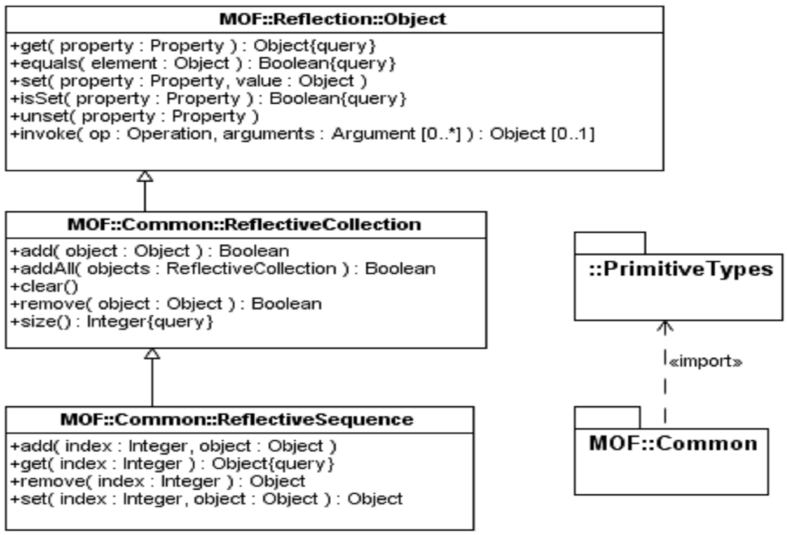

Figure 10.2 - The Common package

10.5 ReflectiveCollection

ReflectiveCollection is a reflective class for accessing properties with more than one possible value. It is defined in package MOF::Common in order to facilitate reuse in many other MOF capabilities.

For ordered properties, ReflectiveSequence (see below) must be returned.

Modifications made to the ReflectiveCollection update the Object's values for that property atomically.

- Exception: throws ClassCastException if the Property's type isInstance(Element) returns false.

add(object: Object): Boolean

Adds object to the last position in the collection. Returns true if the object was added.

addAll(elements: ReflectiveSequence): Boolean

Adds the objects to the end of the collection. Returns true if any elements were added.

clear()

Removes all objects from the collection.

remove(object: Object): Boolean

Removes the specified object from the collection. Returns true if the object was removed.

size(): Integer

Returns the number of objects in the collection.

10.6 ReflectiveSequence

ReflectiveSequence is a subclass of ReflectiveCollection that is used for accessing ordered properties with more than one possible value. Modifications made to the ReflectiveSequence update the Element's values for that property atomically.

- Exception: throws IllegalArgumentException if a duplicate would be added to the collection and Property.isUnique()==true.
- Exception: throws IndexOutOfBoundsException if an index out of the range of 0 <= index < size() is used.
- Exception: throws IllegalArgumentException if a duplicate would be added to the list and Property is of type Class or Property.isUnique()==true.

add(index: Integer, object: Object)

Adds object to the specified index in the sequence, shifting later objects.

get(index: Integer): Object

Returns the object at the given index in the sequence.

remove(index: Integer): Object

Removes the object at the specified index from the sequence. Returns the object removed.

set(index: Integer, object: Object): Object

Replaces the object at the specified index with the new object. The removed object is returned.
Behavior of particular operations defined in ReflectiveCollection is the following when applied to a ReflectiveSequence:

add(object: Object): Boolean

Adds object to the end of the sequence. Returns true if the object was added.

addAll(objects: ReflectiveCollection): Boolean

Adds any objects from the parameter collection to the end of the target sequence:

- in the same order if parameter is unordered then the ordering is random.

- if the target is a unique sequence, then only if they are not already present, this includes objects already added from the parameter that has the effect of removing duplicates from the parameter collection if not unique.

Returns true if any objects were added.

remove(object: Object): Boolean

Removes the first occurrence of the specified object from the sequence.

11 Extension

11.1 General

MOF models provide the ability to define metamodel elements like classes that have properties and operations. However, it is sometimes necessary to dynamically annotate model elements with additional, perhaps unanticipated, information. This information could include information missing from the model, or data required by a particular tool. The MOF Extension capability provides a simple mechanism to associate a collection of name-value pairs with model elements in order to address this need. This is shown in Figure 11.1.

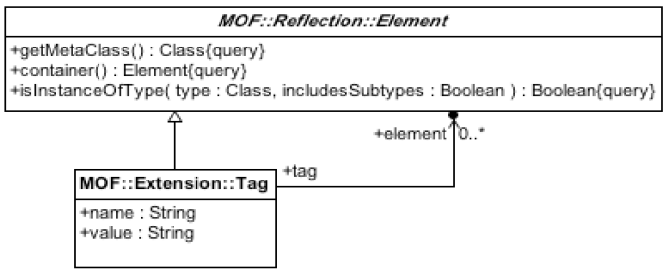

Figure 11.1 - The Extension package

11.2 Tag

A Tag represents a single piece of information that can be associated with any number of model elements. A model element can be associated with many Tags, and the same Tag can be associated with many model elements.

Properties

- name: String - The name used to distinguish Tags associated with a model element.
- value: String - The value of the Tag. MOF places no meaning on these values.
- elements: Element [0..*] - The elements that tag is applied to.
- owner:Element [0..1] - The element that owns the tag (for management purposes).

Operations

No additional operations.

Constraints

No additional constraints.

Semantics

A Tag represents a named value that can be associated with zero or more model elements. A model element cannot have more than one tag with the same name. How tags for a model element are located is not specified by MOF.

A Tag may be owned by another element. This might be a Package, where the tags have been applied externally to the model, or one of the Elements to which the Tag has been applied.

Rationale

Simple string name-value pairs provide extensibility for MOF models that cover a broad range of requirements. They are included to reduce the need to redefine metamodels in order to provide simple, dynamic extensions.

12 The Essential MOF (EMOF) Model

12.1 General

This clause defines Essential MOF, which is the subset of MOF that closely corresponds to the facilities found in OOPLs and XML. The value of Essential MOF is that it provides a straightforward framework for mapping MOF models to implementations such as JMI and XMI for simple metamodels. A primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions (by the usual class extension mechanism in MOF) for more sophisticated metamodeling using CMOF. Both EMOF and CMOF (defined in the next clause) reuse the UML metamodel. The motivation behind this goal is to lower the barrier to entry for model driven tool development and tool integration.

The EMOF Model uses constrained UML 2 class models and includes additional language capabilities defined in this International Standard. As shown in Figure 12-1, the EMOF model merges the Reflection, Identifiers, and Extension capability packages to provide services for discovering, manipulating, identifying, and extending metadata. Package MOF::Common is merged also to provide MOF-internal features.

EMOF, like all metamodels in the MOF 2 and UML 2 family, is described as a UML model. However, full support of EMOF requires it to be specified in itself, removing any package merge and redefinitions that may have been specified in the UML model. This clause provides the UML model of EMOF, and the complete, merged EMOF model. This results in a complete, standalone model of EMOF that has no dependencies on any other packages, or metamodeling capabilities that are not supported by EMOF itself.

NOTE: The abstract semantics specified in “CMOF Abstract Semantics” are optional for EMOF.

The relationship between EMOF and the UML metamodel requires further explanation. EMOF merges UML with the MOF capabilities and a few extensions of its own that are described below. Ideally, EMOF would just extend UML using subclasses that provide additional properties and operations. Then EMOF could be formally specified in EMOF without requiring package merge. However, this is not sufficient because Reflection has to introduce Object in the class hierarchy as a new superclass of UML::Element that requires the merge. As a result of the merge, EMOF is a separate model that merges UML, but does not inherit from it.

By using PackageMerge, EMOF is directly compatible with UML XMI files. Defining EMOF using package merge also ensures EMOF will get updated with any changes to Basic UML. The reason for specifying the complete, merged EMOF model in this clause is to provide a metamodel that can be used to bootstrap metamodel tools rooted in EMOF without requiring an implementation of CMOF and package merge semantics.

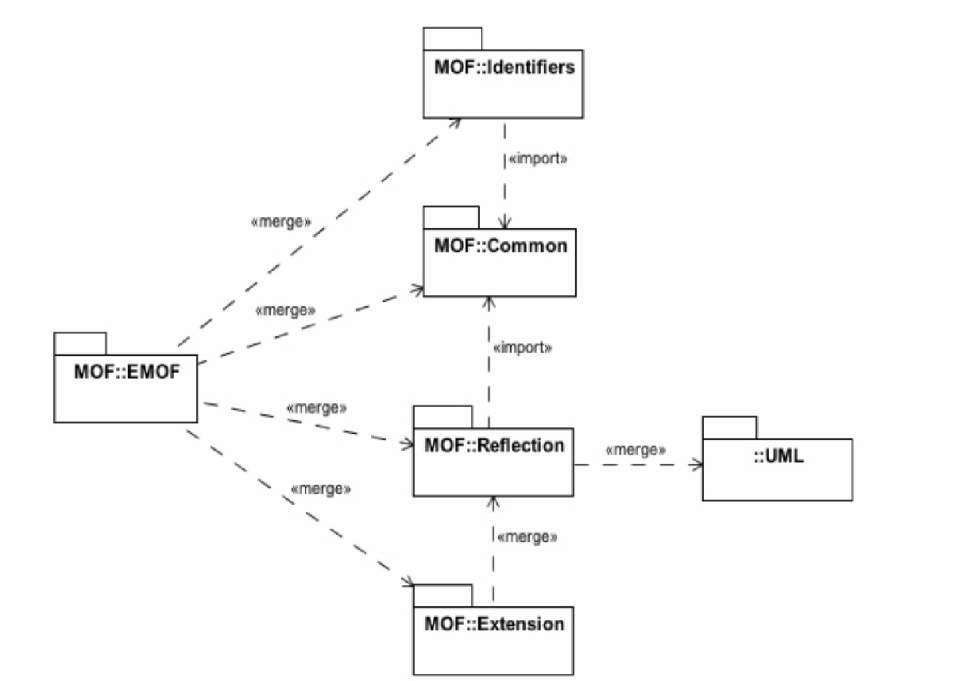

Figure 12.1- EMOF Model - Overview

EMOF makes use of the minimal set of elements required to model object-oriented systems. EMOF reuses the UML metamodel as is for metamodel structure without any extensions, although it does introduce some constraints.

12.2 EMOF Merged Model

This sub clause provides the equivalent EMOF model, obtained by merge with UML the MOF capabilities and omitting those classes and properties excluded, or required to be empty, by EMOF constraints. Thus it represents those parts of UML that can be used to specify EMOF metamodels. It is completely specified in EMOF itself after applying the package merge semantics. The description of the model elements is identical to that found in UML and is not repeated here.

The results of merging the capabilities described in the next sub clause are also shown in some of the diagrams.

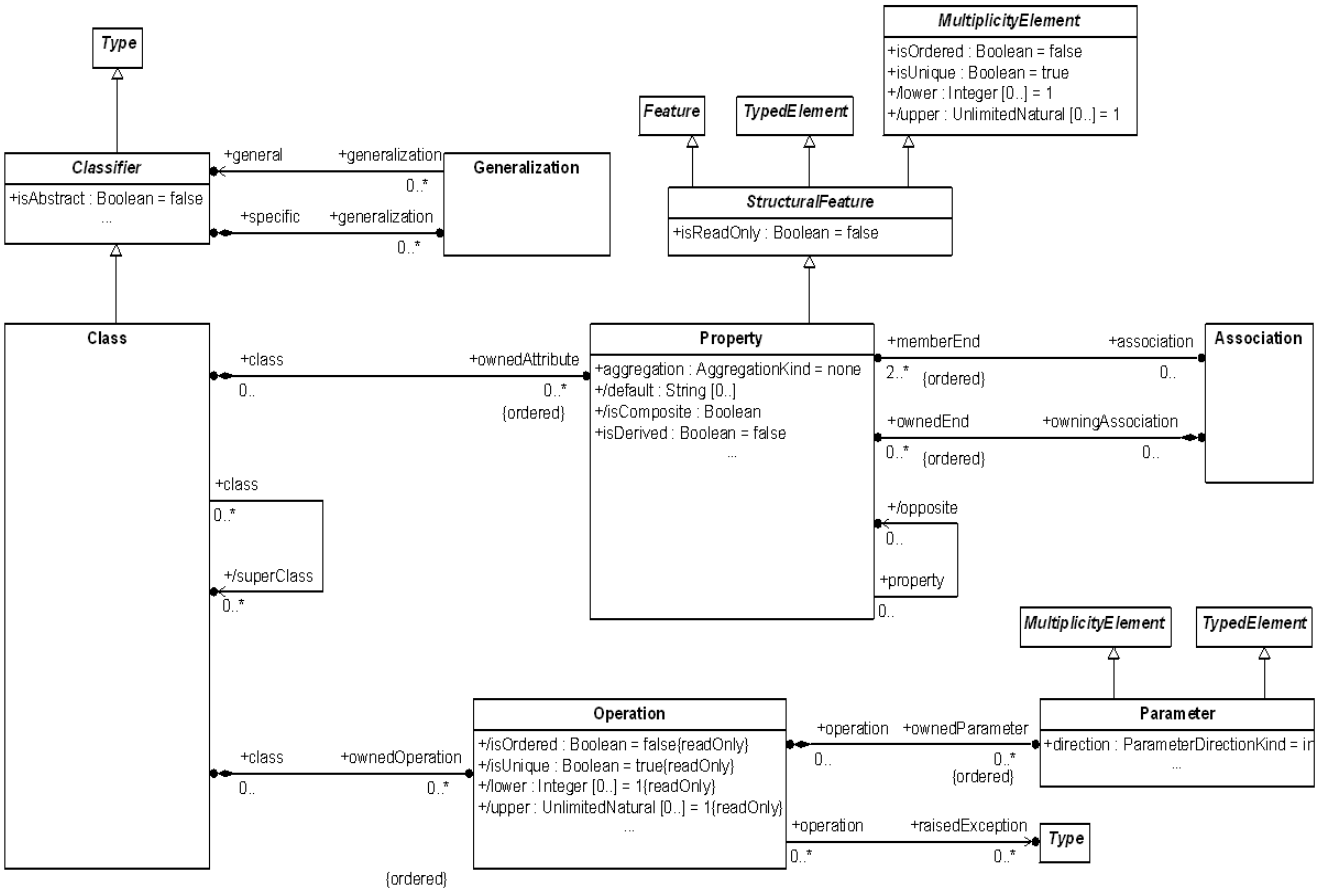

Figure 12.2 - EMOF Classes

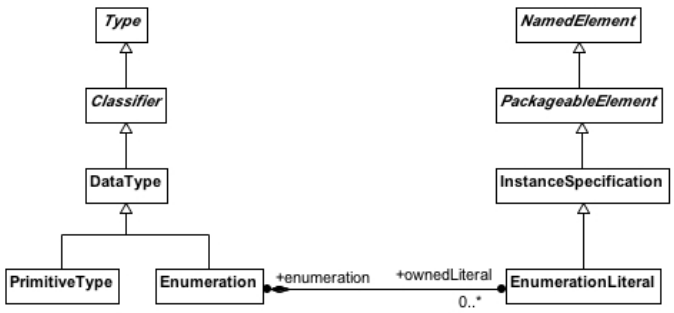

Figure 12.3 - EMOF Data Types

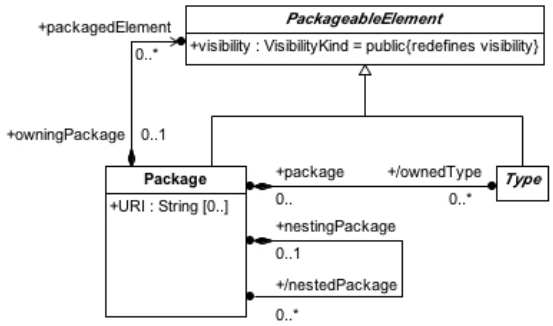

Figure 12.4 - EMOF Package

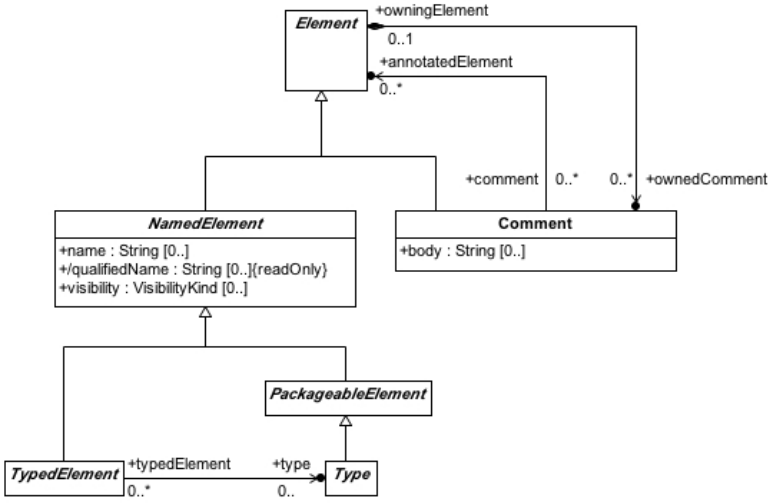

Figure 12.5 - EMOF Types

12.3 Merged Elements from MOF

The EMOF Model merges the following packages from MOF. See the capabilities clauses (9 through 11) for diagrams of the EMOF capabilities.

- Identifiers
- Reflection
- PrimitiveTypes
- Extensions

12.4 EMOF Constraints

These constraints have a formal representation in executable OCL, as referenced from Annex B.

[1] The type of Operation::raisedException is limited to be Class rather than Type.

[2] Notationally, the option is disallowed of suppressing navigation arrows such that bidirectional associations are indistinguishable from non-navigable associations.

[3] Names are required for all NamedElements except for ValueSpecifications.

[4] Core::Basic and EMOF does not support visibilities. All property visibilities must be explicitly set to public where applicable, that is for all NamedElements, ElementImports and PackageImports. Furthermore, no alias is allowed for any ElementImport.

[5] The definitions of Boolean, Integer, and String are consistent with the following implementation definitions: o Boolean: http://www.w3.org/TR/xmlschema-2/#boolean o Integer: http://www.w3.org/TR/xmlschema-2/#integer o String: http://www.w3.org/TR/xmlschema-2/#string [XSD-D].

[6] All the abstract semantics specified in the Clause 15, “CMOF Abstract Semantics” are optional for EMOF.

[7] Property.isID can only be true for one Property of a Class.

[8] An EMOF metamodel is restricted to use the following concrete metaclasses from UML’s Kernel:

- Association
- Class
- Comment
- DataType
- Enumeration
- EnumerationLiteral
- Generalization
- InstanceSpecification
- InstanceValue
- LiteralBoolean
- LiteralInteger
- LiteralNull
- LiteralReal
- LiteralString
- LiteralUnlimitedNatural
- Operation
- Package
- Parameter
- PrimitiveType
- Property
- Slot

[9] The following properties must be empty:

- Association::navigableOwnedEnd

- Class::nestedClassifier
- Classifier::/general for instances of Datatype
- Operation::bodyCondition
- Operation::postcondition
- Operation::precondition
- Operation::redefinedOperation
- Parameter::defaultValue
- Property::qualifier
- Property::redefinedProperty
- Property::subsettedProperty

[10] The following properties must be false:

- Association::isDerived
- Classifier::isFinalSpecialization
- Feature::isStatic
- Property::isDerivedUnion
- RedefinableElement::isLeaf

[11] Generalization::isSubstitutable must be true.

[12] An Association has exactly 2 memberEnds, may never have a navigableOwnedEnd (they will always be owned by Classes) and may have at most one ownedEnd.

[13] An Operation can have up to one Parameter whose direction is ‘return;’ furthermore, an Operation cannot have any ParameterSet per constraint [8].

[14] Comments may only annotate instances of NamedElement.

[15] Only one member attribute of a Class may have isId=true.

[16] Property::aggregation must be either ‘none’ or ‘composite.’

[17] Enumerations may not have attributes or operations.

[18] BehavioralFeature must be sequential.

[19] Class must not be active.

[20] An EnumerationLiteral must not have a ValueSpecification.

[21] An Operation Parameter must have no effect, exception, or streaming characteristics.

[22] A TypedElement cannot be typed by an Association.

[23] A TypedElement other than a LiteralSpecification or an OpaqueExpression must have a Type.

[24] A TypedElement that is a kind of Parameter or Property typed by a Class cannot have a default value.

[25] For a TypedElement that is a kind of Parameter or Property typed by an Enumeration, the defaultValue, if any, must be a kind of InstanceValue.

[26] For a TypedElement that is a kind of Parameter or Property typed by a PrimitiveType, the defaultValue, if any, must be a kind of LiteralSpecification.

[27] A composite subsetting Property with mandatory multiplicity cannot subset another composite Property with mandatory multiplicity.

[28] A Property typed by a kind of DataType must have aggregation = none.

[29] A Property owned by a DataType can only be typed by a DataType.

[30] Each Association memberEnd Property must be typed by a Class.

[31] A multi-valued Property or Parameter cannot have a default value.

[32] The values of MultiplicityElement::lowerValue and upperValue must be of kind LiteralInteger and LiteralUnlimitedNatural respectively.

12.5 EMOF Definitions and Usage Guidelines for the UML Models

When the EMOF package is used for metadata management the following usage rules apply.

Package

- Although EMOF defines Package and nested packages, EMOF always refers to model elements by direct object reference. EMOF never uses any of the names of the elements. There are no operations to access anything by NamedElement::name. Instances of EMOF models may provide additional namespace semantics to nested packages as needed.

Properties

- All properties are modified atomically.
- When a value is updated, the old value is no longer referred to.
- Derived properties are updated when accessed or when their derived source changes as determined by the implementation. They may also be updated specifically using set() if they are updateable.

Type==DataType

- The value of a Property is the default when an object is created or when the property is unset.
- Properties of multiplicity upper bound > 1 have empty lists to indicate no values are set. Values of the list are unique if Property.isUnique==true.
- “Identifier” properties are properties having property.idID==true.

Type==Class

- Properties of multiplicity upper bound == 1 have value null to indicate no object is referenced.
- Properties of multiplicity upper bound > 1 have empty lists to indicate no objects are referenced. Null is not a valid value within the list.
- EMOF does not use the names of the properties, the access is by the Property argument of the reflective interfaces. It does not matter what the names of the Properties are, the names are never used in EMOF. There is no special meaning for having similar names. The same is true for operations, there is no use of the names, and there is no name collision, override, or redefinition semantics. EMOF does not have an invoke method as part of the reflective interface, so there are no semantics for calling an EMOF operation. The names and types of parameters are never compared and there is no restriction on what they can have singly or in combination. Other instances of EMOF metamodels, or language mappings, such as JMI2, may have additional semantics or find that there are practical restrictions requiring more specific definitions of the meaning of inheritance.

Property::isComposite==true

- An object may have only one container.
- Container properties are always multiplicity upper bound 1.
- Only one container property may be non-null.
- Cyclic containment is invalid.
- If an object has an existing container and a new container is to be set, the object is removed from the old container before the new container is set.
- Adding a container updates both the container and containment properties on the contained and containing objects, respectively. The opposite end is updated first.
- The new value is added to this property.

Property::isComposite==false, Bidirectional

- The object is first removed from the opposite end of the property.
- If the new value's opposite property is of multiplicity upper bound == 1, its old value is removed.
- This object is added to the new value's opposite property.
- The new value is added to this property.

Object

- Everything that may be accessed by MOF is an Object.
- An Object that is not also an Element may be an instance of one DataType.

12.6 Predefined Tags

This sub clause defines a predefined Tag whose name is "org.omg.emof.oppositeRoleName" that can be applied to instances of Property within instances of the EMOF model.

Constraints

context Tag inv:

The predefined Tag can only be applied to instances of Property whose "opposite" Property is empty

name = "org.omg.emof.oppositeRoleName" implies

element.oclIsKindOf(Property) and element.oclAsType(Property).opposite->isEmpty()

Semantics

If an instance of a Tag has "org.omg.emof.oppositeRoleName" as its "name," then its "value" specifies a role name that expressions can use to traverse in the opposite direction of the Property, such as OCL expressions and QVT expressions.

If an expression uses a role name specified using a Tag with “name” “org.omg.emof.oppositeRoleName,” and more than one Property has such a Tag with that role name, then it is up to the expression language to decide whether this is an error condition or represents a reverse navigation across all those Properties. An expression language should not choose to pick one such Property at random in case of ambiguity.

Rationale

Use of this Tag is lighter weight than using Property’s “opposite” Property. Use of the “opposite” Property in all cases where what is required is only the ability for expressions to traverse in the opposite direction would have the following negative consequences:

- It would result in tighter coupling among Classes.
- It would add to the runtime burden that instances of the model place upon the underlying infrastructure that manages them, by: 1) increasing the overall footprint, since the opposite Property adds substantially to the contract of the Class that owns the additional Property designated as the opposite of the original Property; 2) requiring that storage be allocated for instances of the additional Property; and 3) requiring that referential integrity be maintained in storage among instances of the original Property and instances of the additional Property.

It is beyond the scope of MOF Core to specify the concrete syntax that expressions use for traversal via the org.omg.emof.oppositeRoleName in languages such as OCL and QVT.

13 CMOF Reflection

13.1 General

CMOF::Reflection provides extended capabilities over the MOF::Reflection package. The package diagram in Figure 13.1 shows how the CMOF::Reflection package extends MOF::Reflection.

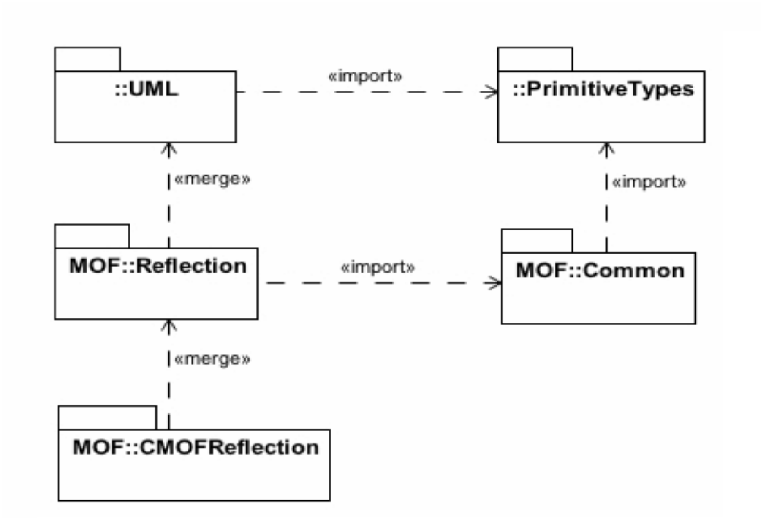

Figure 13.1 - CMOF Reflection

CMOF::Reflection merges additional operations into the existing Object, Extent, and Factory classes, and adds a Link class and an Argument datatype. These additions by CMOF::Reflection are shown in Figure 13.2.

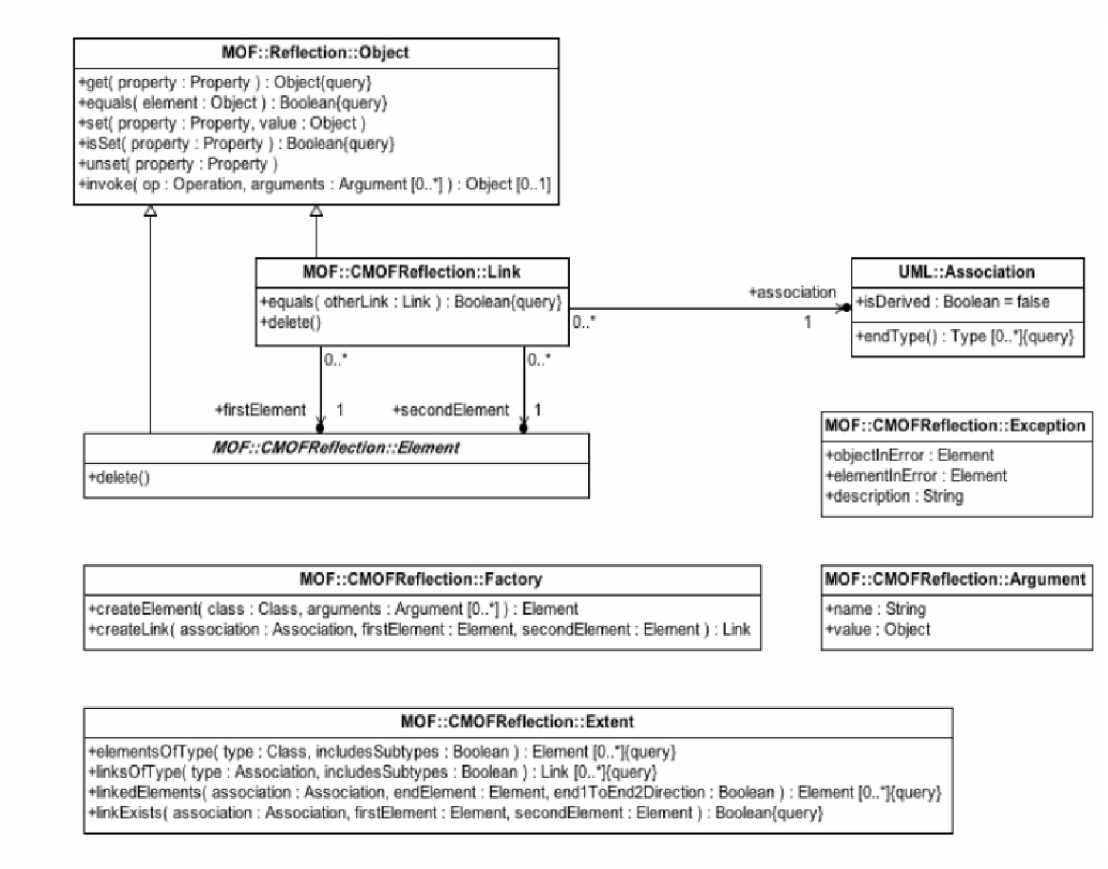

Figure 13.2 - CMOF Reflection package

13.2 Link

This is a new class that represents an instance of an Association, in the same way that Element represents an instance of a Class.

Properties

association: Association - This is the Association of which the Link is an instance.

firstElement: Element - This is the Element associated with the first end of the Association.

secondElement: Element - This is the Element associated with the second end of the Association.

Operations

equals(otherLink:Link): Boolean

Returns True if the otherLink has association, firstElement, secondElement all equal to those on this Link.

delete()

Deletes the Link. This may leave the same elements associated by other links for this Association.

Constraints

The firstElement must conform to the type of the first memberEnd of the association.

The secondElement must conform to the type of the second memberEnd of the association.

The set of Links as a whole must not break the multiplicity constraints of the association member ends.

Semantics

When a Link is created, it is not inserted into any Extent.

When one or more ends of the Association are ordered, links carry ordering information in addition to their end values.

Rationale

Since MOF 2 allows the same pair of elements to be linked more than once in the same Association (if isUnique=false for the association ends), then Link needs to be more a simple tuple value, though not as heavyweight as a first class Element.

13.3 Argument

This is a new datatype that is used to represent named arguments to open-ended reflective operations. It is open-ended and allows both Elements and data values to be supplied.

Properties

name: String - The name of the argument.

value: Object - The value of the argument.

Constraints

Argument is a data type supporting open reflective operations. As a data type, it has no semantics of its own. Constraints will be dependent on the context of where the Argument is supplied.

Semantics

None.

Rationale

Since MOF 2 allows Operation parameters and Properties to have defaults, it is necessary to explicitly identify the values supplied.

13.4 Object

CMOF Reflection adds the following extra operations.

invoke(op:Operation, arguments : Argument[0..*]) : Object[0..*]

Calls the supplied Operation on the object, passing the supplied Arguments and returning the result. If the operation produces more than one result value, then the result of the invoke operation is a kind of ReflectiveCollection containing all of the result values produced.

The Operation must be defined on the Class of the Object, and the arguments must refer to Parameters of the Operation. If an Argument is not supplied for a Parameter, its default value, if any, will be used.

Rationale

Adds the equivalent of MOF 1.4 capabilities.

13.5 Element

CMOF Reflection adds the following extra operations.

Operations

delete()

Deletes the Element.

isInstanceOfType(type: Class, includeSubtypes: Boolean): Boolean

Returns true if this element is an instance of the specified Class, or if the includeSubtypes is true, any of its subclasses.

Rationale

Adds the equivalent of MOF 1.4 capabilities.

13.6 Factory

CMOF Reflection adds two extra operations.

Operations

createElement(class:Class, arguments : Argument[0..*]) : Element

Unlike the simple create() operation this allows arguments to be provided for use as the initial values of properties.

The arguments must refer to DataType Properties of the Class. If an Argument is not supplied for a Property its default value, if any, will be used.

createLink(association : Association, firstElement : Object, secondElement : Object) : Link

This creates a Link from 2 supplied Elements that is an instance of the supplied Association. The firstElement is associated with the first end (the properties comprising the association ends are ordered) and must conform to its type. And correspondingly for the secondElement.

Rationale

Adds the equivalent of MOF 1.4 capabilities.

13.7 Extent

CMOF Reflection adds four extra operations.

Operations

elementsOfType(type : Class, includeSubtypes : Boolean) : Element[0..*]

This returns those elements in the extent that are instances of the supplied Class. If includeSubtypes is true, then instances of any subclasses are also returned.

linksOfType(type : Association, includesSubtypes : Boolean) : Link[0..*]

This returns those links in the extent that are instances of the supplied Association, or of any of its subclasses if includesSybtypes is true.

linkedElements(association : Association, endElement : Element, end1ToEnd2Direction : Boolean) : Element[0..*]

This navigates the supplied Association from the supplied Element. The direction of navigation is given by the end1ToEnd2Direction parameter: if true, then the supplied Element is treated as the first end of the Association.

linkExists(association : Association, firstElement : Element, secondElement : Element): Boolean

This returns true if there exists at least one link for the association between the supplied elements at their respective ends.

14 The Complete MOF (CMOF) Model

14.1 General

The CMOF Model is the metamodel used to specify other metamodels such as UML2. It is built from EMOF and selected elements of the UML metamodel. The Model package does not define any classes of its own. Rather, it merges packages with its extensions that together define basic metamodeling capabilities. The complete Package structure constituting CMOF is shown in Figure 14.1.

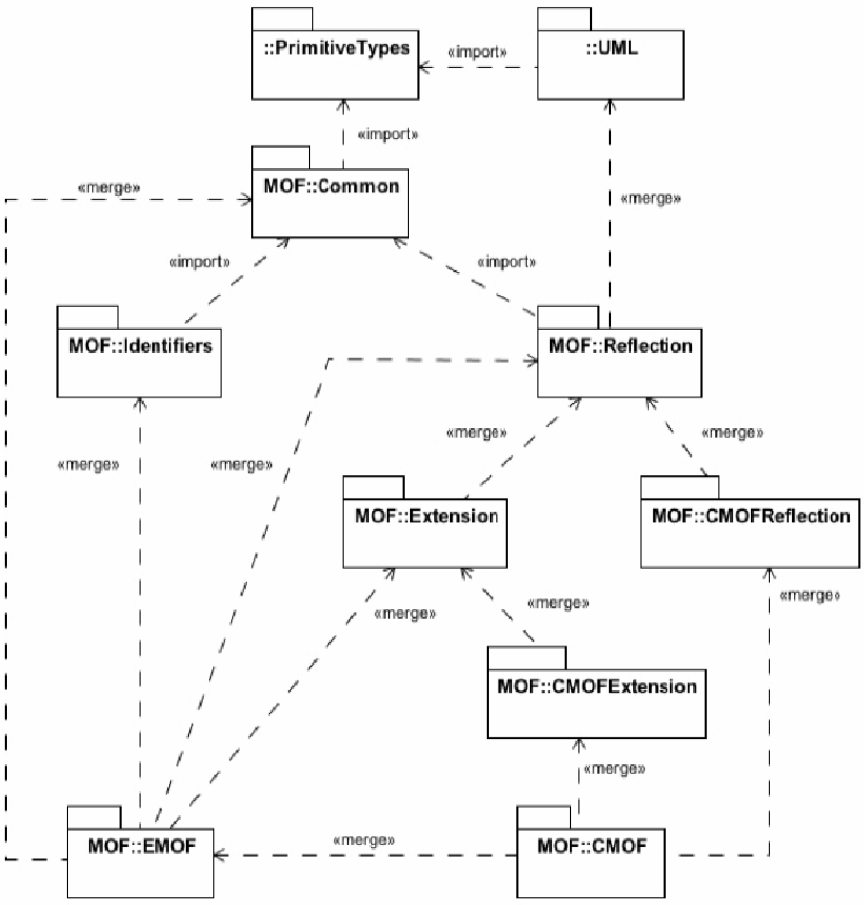

Figure 14.1 - CMOF Packages

14.2 Elements used from UML 2

Figure 14.2 shows some of the key concrete classes and associations of UML 2 that are used to specify CMOF metamodels. There are many other important elements that are used from UML, but these provide the structure of class modeling. See the UML specification for full details.

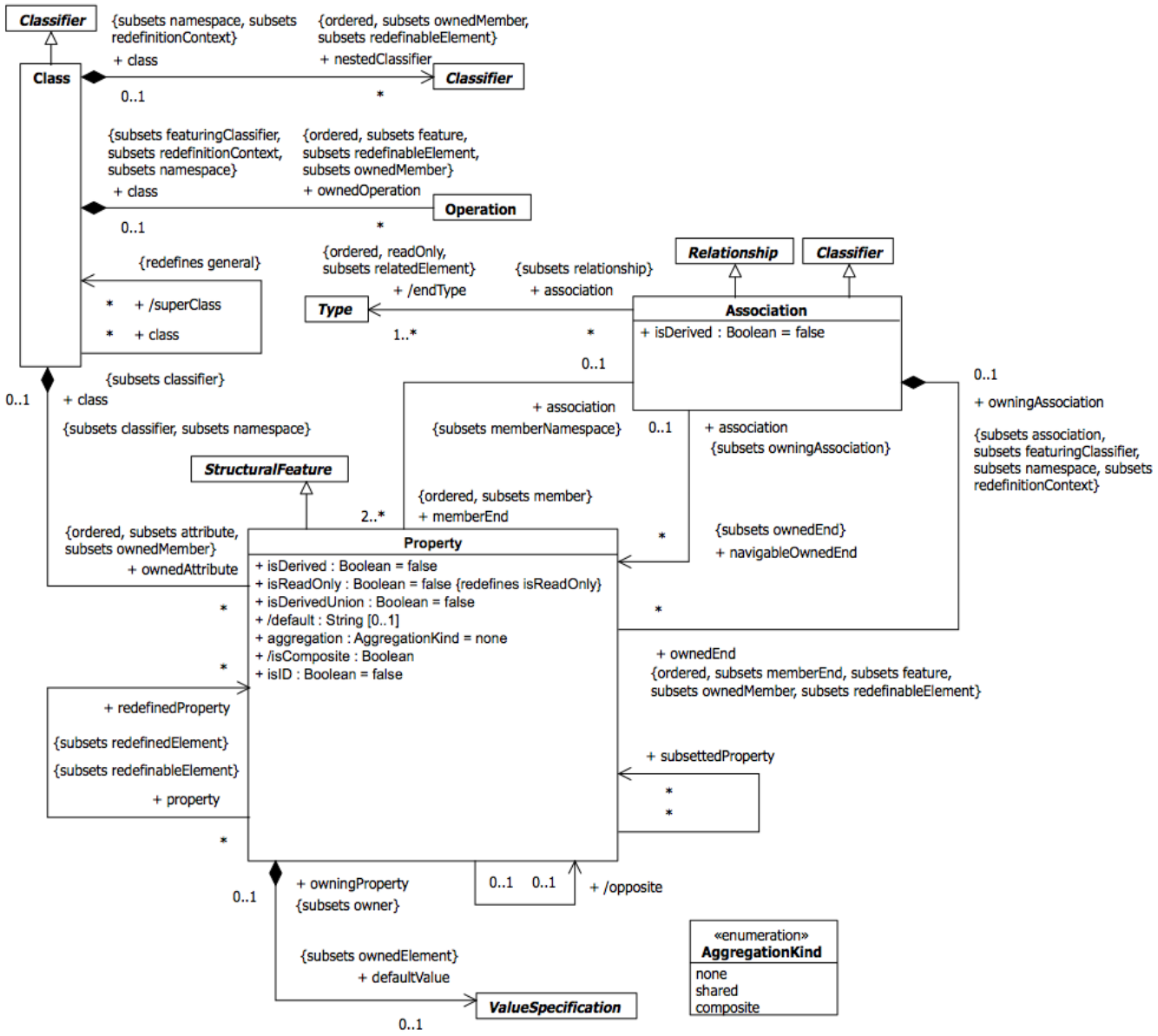

Figure 14.2 - Key concrete classes from UML2 Kernel package

14.3 Imported Elements from MOF

The CMOF Model merges package EMOF from MOF, which includes MOF capabilities packages:

- Identifiers
- Reflection
- Extension

14.4 CMOF Constraints

This sub clause details the constraints owned by the CMOF package that are applied to metamodels to be processed by a CMOF implementation. These constraints supersede the EMOF constraints from 12.4; that is, validating a CMOF metamodel should be done with respect to all the CMOF constraints defined in this clause ignoring all the constraint definitions from 12.4.

The CMOF metamodel, other MOF packages, and UML itself conform to all of these.

These constraints have a formal representation in executable OCL, as referenced in Annex B.

[1] The multiplicity of Association::memberEnd is limited to 2 rather than 2..* (i.e., n-ary Associations are not supported); unlike EMOF, CMOF associations can have navigable association-owned ends.

[2] The type of Operation::raisedException is limited to be Class rather than Type.

[3] In order to support limited implementations of the Integer class, each instance of Integer occurring as an attribute value of an element is in the range of integers that can be represented using a 32-bit two's complement format. In other words, each integer used is in the range from -2_{31} through $2_{31} - 1$.

[4] In order to support limited implementations of the String class, each instance of String occurring as an attribute value of an element has a length that does not exceed 65535 characters.

[5] Notationally, the option is disallowed of suppressing navigation arrows such that bidirectional associations are indistinguishable from non-navigable associations.

[6] Names are required for all NamedElements except for ValueSpecifications.

[7] CMOF does not support visibilities. All property visibilities must be explicitly set to public where applicable, that is for all NamedElements, ElementImports, and PackageImports. Furthermore, no alias is allowed for any ElementImport.

[8] Enumerations may not have attributes or operations.

[9] Property.isID can only be true for one Property of a Class.

[10] A CMOF metamodel is restricted to use the following concrete metaclasses from UML:

- Association
- Class
- Comment
- Constraint
- DataType
- ElementImport
- Enumeration
- EnumerationLiteral

- Generalization
- InstanceSpecification
- InstanceValue
- LiteralBoolean
- LiteralInteger
- LiteralNull
- LiteralReal
- LiteralString
- LiteralUnlimitedNatural
- OpaqueExpression
- Operation
- Package
- PackageImport
- PackageMerge
- Parameter
- PrimitiveType
- Property
- Slot

[11] The following properties must be empty:

- Class::nestedClassifier
- Property::qualifier

[12] The value of Feature::isStatic must be false.

[13] A multi-valued Property or Parameter cannot have a default value. The default value of a Property or Parameter typed by a PrimitiveType must be a kind of LiteralSpecification. The default value of a Property or Parameter typed by an Enumeration must be a kind of InstanceValue. A Property or Parameter typed by a Class cannot have a default value.

[14] The values of MultiplicityElement::lowerValue and upperValue must be of kind LiteralInteger and LiteralUnlimitedNatural respectively.

[15] Generalization::isSubstitutable must be true

[16] Only one member attribute of a Class may have isId=true. Any others (e.g., those inherited) must be redefined: either made unavailable or redefined to change isId = false.

[17] Property::aggregation must be either ‘none’ or ‘composite.’

[18] BehavioralFeature must be sequential.

[19] Class must not be active.

[20] An EnumerationLiteral must not have a ValueSpecification.

[21] An Operation Parameter must have no effect, exception, or streaming characteristics.

[22] A TypedElement cannot be typed by an Association.

[23] A TypedElement other than a LiteralSpecification or an OpaqueExpression must have a Type.

[24] A TypedElement that is a kind of Parameter or Property typed by a Class cannot have a default value.

[25] For a TypedElement that is a kind of Parameter or Property typed by an Enumeration, the defaultValue, if any, must be a kind of InstanceValue.

[26] For a TypedElement that is a kind of Parameter or Property typed by an PrimitiveType, the defaultValue, if any, must be a kind of LiteralSpecification.

[27] A composite subsetting Property with mandatory multiplicity cannot subset another composite Property with mandatory multiplicity.

[28] A Property typed by a kind of DataType must have aggregation = none.

[29] A Property owned by a DataType can only be typed by a DataType.

[30] Each Association memberEnd Property must be typed by a Class.

[31] A Constraint must constrain at least one element and must be specified via an OpaqueExpression.

[32] The body of an OpaqueExpression must not be empty.

14.5 CMOF Extensions to Capabilities

This sub clause details CMOF extensions to the MOF2 capabilities.

14.5.1 Reflection

[1] CMOF extends Factory to allow the format of the string argument of Factory::createFromString and the result of Factory::convertToString to be specified as defined in "Meta Object Facility (MOF) 2.0 XMI Mapping" in order to support default values for structured data types.

14.5.2 Extension

[1] CMOF extends package Extension with an association between Element with role tagOwner and Tag with a navigable association-owned end role ownedTag.

Associations:

tagOwner : Element[0..1] {subsets Element::owner}

NOTE: Although the ownedTag end is owned by the association, it must be navigable to support retrieving the Tags that an object owns using UML's ReadLinkAction (see UML2.4, sub clause 11.3.33). Additionally, OCL provides the capability to retrieve the Tags that an Element, e, owns, with the expression: e.ownedTag (see OCL2.2, sub clause 7.5.3). That is, the following invariant follows from Figure 14.3.

```
context Element
            inv CMOF_ElementTag: ownedTag->forAll(tagOwner = self)
```

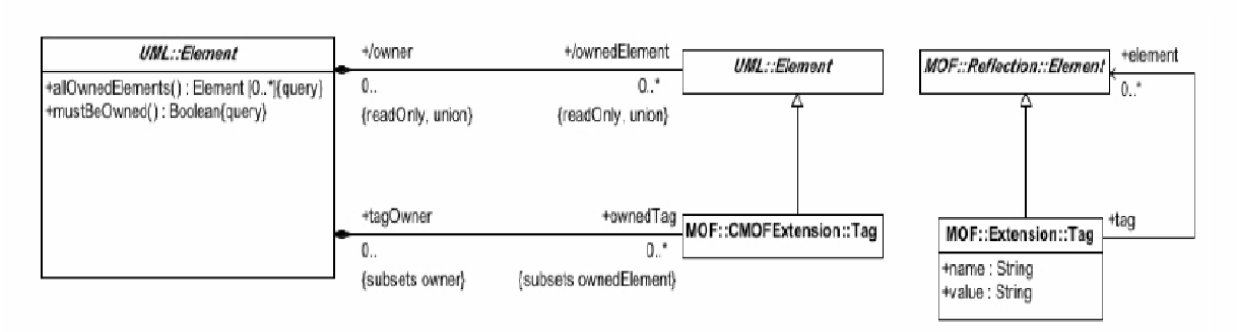

Figure 14.3 - CMOF Extension Package

15 CMOF Abstract Semantics

15.1 General

This clause describes the semantics of CMOF by describing the functional capabilities of a modeled system and how those capabilities are related to elements in the model. These capabilities are independent of any mapping to implementation technology, so their descriptions are abstract.

Note that all of these capabilities are limited by the types, multiplicities, constraints, visibility, setability, etc. imposed by the model, and are possibly further constrained by other considerations such as access control security, immutability of historical versions of data, etc. Therefore, use of these capabilities can fail in specific situations - a failure results in an exception being raised (see Exceptions package).

15.2 Approach

MOF is a platform-independent metadata management framework that involves creating, manipulating, finding, changing, and destroying objects and relationships between those objects as prescribed by metamodels. This sub clause describes the Core capabilities that MOF provides, and the semantics and behaviors of those activities. These capabilities may be extended or refined in the further specifications in the MOF 2 series. It is not the intention of this sub clause to mandate that all MOF implementations need to support all of these capabilities: more to define what the capabilities mean when they are provided. Compliance points are described separately. For example, this sub clause defines the semantics of Reflection: this constrains those implementations that provide Reflection but does not require all implementations to provide it.

For these capabilities to be well-defined, well-behaved, and understandable, some of these capabilities are described with respect to a lightweight notion of logical “extents” of model elements that provide some abstract notion of location and context. These will be more fully defined as part of the MOF 2 Facility and Object Lifecycle specification.

The goal of defining these capabilities is that they provide a single platform-independent definition that can be used as the basis of the language bindings in order to gain some level of consistency and interoperability. It also allows the semantics and constraints resulting from metamodeling decisions to be defined: increasing the level of semantic definition.

This sub clause takes MOF from being a meta-metamodel to being a modeled system specification (a Platform Independent Model). This of necessity introduces more detail and constraints than present in the UML metamodel on which it is based. In particular it has been necessary to extend the UML2 Instances model to achieve this.

Though the approach is described in terms of the instances model (e.g., Slots) it is important to stress that this is a specification model and does not determine an implementation approach. Instances classes such as Slot do not appear in the operations specified: implementations should behave as if they were implemented using Slots but will in general be implemented using far more efficient mechanisms.

The specification is in terms of the reflective interface but it is intended that there be specific interfaces generated for specific metamodels.

15.3 MOF Instances Model

Principles

This semantic domain just covers the Classes Diagram from Constructs.

In general the approach is to avoid redundancy and only to have Slots where needed (e.g., not for derived attributes). This is to simplify specification of update behavior (not attempted here) and potentially the specification of XMI serialization.

The exception is that association instances are represented both as AssociationInstances and via Slots on the linked objects (for navigable association ends only). In theory the navigable end values could be derived via queries over the AssociationInstances but this was not done:

- for simplicity of explanation.
- to retain the 'illusion' that these properties are true attributes.
- to provide for greater consistency with Basic.

Datavalues act as both Instances (since they may have slots) and as ValueSpecifications: since datavalues are always considered directly stored in a slot rather than being referred to (which would require some sort of identity).

Figure 15.1 represents the Semantic Domain model for Constructs. It extends the abstract syntax for Instances.

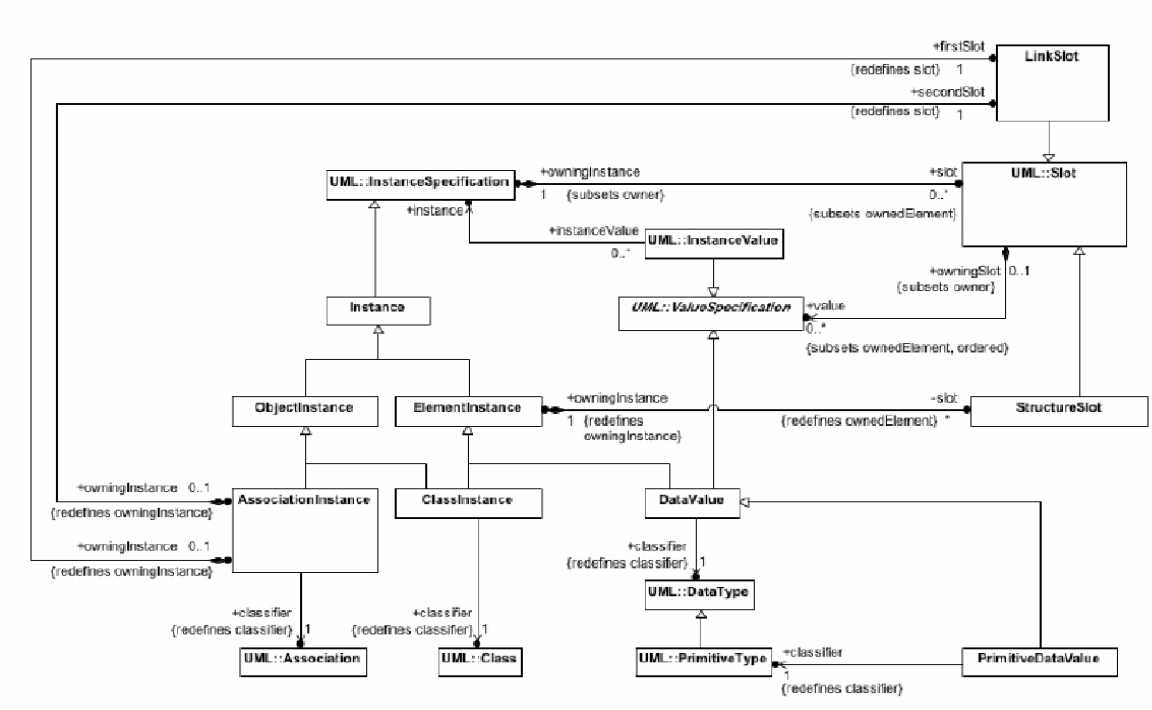

Figure 15.1 - Semantic Domain model for Constructs

The following represent constraints on the classes introduced.

ObjectInstance (applies to both ClassInstances and AssociationInstances)

1. There is exactly one slot for each stored StructuralFeature and no further slots. The storedStructuralFeatures include inherited ones but exclude:
   - Derived properties (including derived unions).
   - Properties that have been redefined (there exists a property that is a member of the classifier that has

redefinedProperty= this).

2. The classifier is not abstract.
3. The instance obeys Constraints that have its classifier as context.

ClassInstance

1. An instance is owned via at most one composition.
2. At most one owning Slot, i.e., a Slot whose property is opposite an isComposite property, may have a value.
3. Compositions are not cyclic.

StructureSlot

1. The number of values corresponds with the multiplicity (upper and lower) of its definingFeature.
2. If the feature isUnique, then no 2 values are equal.
3. Slots for opposite properties tie up. For all values in slot the referenced objects do refer back via the opposite property; moreover no Slot of the opposite property in other objects refers to the owner of this Slot.
4. The slot's values are a subset of those for each slot it subsets.

LinkSlot

1. Where the feature is a navigable end, then the ClassInstance Slot is consistent with the Link slot.
2. The number of links is consistent with the multiplicity and uniqueness of the ends.

PrimitiveDataValue

1. If classifier is an Enumeration, then the valueRepresentation is the name of a valid EnumerationLiteral.

Further constraints on Abstract Syntax

The following represent new constraints that should be introduced.

Datatype

For all properties, isReadOnly is true, isComposite is false, isDerivedUnion is false.
Datatypes may not participate in Associations.

PrimitiveType

For all properties, isDerived is true.

Enumeration

For all properties, isDerived is true.

Property

If one of a pair of opposites isUnique, then so must the other be. At most one of redefinedProperty and subsettedProperty may be set.

Association

An Association is derived if all its Properties are derived.

15.4 Remarks on MOF Instance Modeling

This sub clause models the reflective capabilities in terms of the Instances model above: so each Reflective signature is interpreted/modeled as the equivalent operation on the Instances model above.

The implementation for derived properties and operations is opaque: inbuild function *extInvoke* is used to call the implementation-supplied code. No slots are allocated for derived properties.

Similarly the evaluation of constraints is deferred to an inbuilt operation *evaluateConstraint*.

An extra function *extent()* is used to represent the current extent or extents of an Object. Its value is context dependent. Moreover it is expected that a Factory will be associated with at least one Extent (this is properly in scope of MOF 2 Facility RFP).

No distinction is made between slots based on multiplicity: it assumed that a slot can hold a collection; also that a collection can be a valid instance of DataValue. In most language bindings it is expected that for a multivalued property (upper bound > 1) that an empty collection will be returned instead of null: for simplicity of specification this is not done here.

Convenience/helper OCL operations are used.

For clarity a distinguished value '*null*' is used here for Property values to indicate they are empty.

15.5 Object Capabilities

Object::getType(): Type modeled as **ObjectInstance::getType(): Type**
post: result = self.classifier

Object::container(): Object modeled as **Instance::container(): ClassInstance**
post: result = self.get(self.owningProperty())

Object::get(Property p): Element
    modeled as **ObjectInstance::get(Property p): ElementInstance**
-- If a foreign association end, then navigate link, else access slot or derive the value
post: (p.namespace.isOclType(Association) and result = navigate(p)) or
    self.propertySlot(p) <> null and (
        (self.propertySlot(p).value <> null and result = self.propertySlot(p).value) or
        result = p.default) or
    (p.isDerivedUnion and result = unionedProperties(p)->union(s| s = self.get(s)) or
    (p.isDerived and result = self.extInvoke('get', p))

Object::set(Property p, Element v)

modeled as ObjectInstance::set(Property p, ElementInstance v)
pre: not(p.isReadOnly)
post: internalSet(p, v)

Object::isSet(Property p): Boolean
modeled as ObjectInstance::isSet(Property p): Boolean
post: result = (self.propertySlot(p).value = null)

Object::unset(Property p) modeled as **ObjectInstance::unset(Property p)**
pre: not(p.isReadOnly)
-- Set to property default - this will have desired effect even if the default is not set (null)
post: internalUnset(p)

Object::delete() modeled as **ObjectInstance::delete()**
-- Delete all composite objects and all slots
post: (self.allProperties->select(p| isComposite(p), delete(self.get(p))) and
self.allSlottableProperties->forAll(p| destroyed(self.propertySlot(p))) and
extent().removeObject(self)
not(extent().objects() includes self) and
destroyed(self)

Object::invoke(Operation op, Set{Tuple{Parameter p, ValueSpecification v}} args):**Element**
modeled as **ClassInstance::invoke(Operation op, Set{Tuple{Parameter p, ValueSpecification v}} args**):**Element**
-- Ensure all supplied parameters are for this operation and the values are of the correct type and all mandatory parameters are supplied

pre: args->forAll(Tuple{p, v}| op.parameter includes p and conformsTo(p.type, v)) and
op.parameter->select(p| p.lower > 1, args includes Tuple{p, x})

Object::isInstanceOfType(type: Class, includeSubclasses: Boolean): **Boolean**
modeled as **ClassInstance::isInstanceOfType(type: Class, includeSubclasses: Boolean**): **Boolean**
post: result = (self.classifier = type or
includeSubclasses and self.classifier.allParents() includes type)

15.6 Link Capabilities

Link:equals(otherLink:Link): Boolean
modeled as **AssociationInstance:equals(otherLink:AssociationInstance): Boolean**

post: result = (self.association = otherLink.association and

self.firstSlot.value = otherLink.firstSlot.value and

self.secondSlot.value = otherLink.secondSlot.value)

Link:delete() modeled as **AssociationInstance:delete()**

post: destroyed(self.firstSlot) and destroyed(self.secondSlot) and

extent().removeObject(self) not(extent().objects() includes self) and

destroyed(self)

15.7 Factory Capabilities

The following are defined on the class Factory:

Factory::createObject(Type t, Set{Tuple{Property p, ValueSpecification v}} args): Object

-- Create the object and slots for properties (including inherited ones) that are not derived and not the redefinition or subset of another; assign the supplied values or default values if any

pre:

-- Check the arguments are valid properties of the correct type and that values are supplied for all mandatory properties with no default

not(isAbstract(t)) and

args->forAll(Tuple{p, v}| t.allProperties includes p and conformsTo(p.type, v)) and

op.parameter->select(p| p.lower > 1 and p.default = null and args includes Tuple{p, x})

-- Create the slots and then set the values from arguments or defaults

post: oclIsNew(result) and

extent().addObject(result)

extent().objects includes result and

result.classifier = c and

t.allSlottableProperties->forAll(a | exists(s:StructureSlot | oclIsNew(s) and

s.definingFeature = a and

s.owningInstance = result) and

t.allProperties->forAll(p| (exists(v | args includes [p,v] and internalSet(p,v))) or

(self.internalUnset(p))

-- also need to cater for setting properties using constraints(?)

Factory::createLink(association : Association, firstObject : Object, secondObject : Object) : Link

modeled as **Factory::createLink(association : Association, firstObject : Object, secondObject : Object) : AssociationInstance**

-- Create the link; assign the supplied objects
pre:
-- Check the objects are valid instances of the correct type
not(association.isAbstract) and conformsTo(association.memberEnd[0].type, firstObject) and conformsTo(associa
tion.memberEnd[1].type, secondObject)
post:
oclIsNew(result) and
extent().addObject(result) and
extent().linksOfType(association) includes result and
result.classifier = association and
oclIsNew(s1) and s1.definingFeature = association.memberEnd[0] and s1.value = firstObject and
oclIsNew(s2) and s2.definingFeature = association.memberEnd[1] and s2.value = secondObject

Factory::createFromString(dataType: DataType, string: String) : Element

modeled as **Factory::createFromString(dataType: DataType, string: String): DataValue**

-- issue: requires a set of syntax rules for literals

pre: self.package.member includes dataType

Factory::convertToString(dataType: DataType, element: Element) : String

modeled as **Factory::createFromString(dataType: DataType, element: DataValue): String**

-- issue: requires a set of syntax rules for literals

pre: self.package.member includes dataType

post: createFromString(dataType, result) = element

-- the string produced should parse to the same value!

15.8 Extent Capabilities

This sub clause describes minimal capabilities for MOF Core. It does not address issues such as extent creation that are in scope of MOF 2 Facility and Object Lifecycle RFP. Neither does it yet cover the use of ‘exclusive’ and ‘useContainment’ attributes: it is assumed that all objects are directly contained in one extent.

As an abstract model of behavior extents are implemented using the following Instance model.

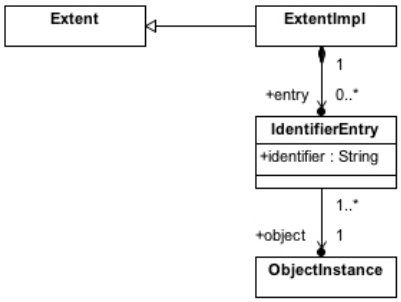

Figure 15.2 - Instance Model for Extent

Extent::objects(): Object modeled as **ExtentImpl:objects(): ObjectInstance**

post: result = entry->object

Extent::objectsOfType(type: Class, includeSubtypes: Boolean): Object modeled as **ExtentImpl:objectsOfType(type:Class, includeSubtypes: Boolean): ObjectInstance**

post:

result = self.entry->object->select(o| o.classifier = type or

(includeSubtypes and o.classifier.allParents includes type)

Extent::linksOfType(type: Association): Link modeled as **linksOfType(type: Association): AssociationInstance**

post:

result = self.entry->object->select(o| o.classifier = type)

Extent::linkedObjects(association : Association, endObject : Object, end1ToEnd2Direction : Boolean) : Object modeled as **ExtentImpl:linkedObjects(association : Association, endObject : Object, end1ToEnd2Direction : Boolean) : ObjectInstance**

post: result = self.entry->object->select(o, r| o.classifier = association and (

(end1ToEnd2Direction and o.firstObject = endObject and r = o.secondObject) or

o.secondObject = endObject and r = o.firstObject))

Extent::linkExists(association : Association, firstObject : Object, secondObject : Object) : Boolean modeled as **Extent::linkExists(association : Association, firstObject : Object, secondObject : Object) : Boolean**

result = self.entry->object->exists(o| o.classifier = association and

o.firstSlot.value = firstObject and

o.secondSlot.value = secondObject)

Extent::identifier(o: Object): String modeled as **ExtentImpl::identifier(o: ObjectInstance): String**

post:

result = self.entry->object->select(eo| eo = o).identifier

Extent::object(id: String): Object modeled as **ExtentImpl::object(id: String): ObjectInstance**

post:

result = self.entry->select(i| i = id).object

15.9 Additional Operations

[1] This gives all of the properties of the class (including inherited) that require a slot: it excludes derived and redefined properties.
Class::allSlottableProperties(): Set(Property);
result = self.allProperties()->select(not is Derived)

[2] All the non-redefined properties (including inherited) of a class.
Class::allProperties(): Set(Property);
result = member->select(oclIsKindOf(Property).oclAsType(Property)

[3] All of the properties directly or indirectly redefined by a property.
Property::allRedefinedProperties() : Set(Property)
result = self.redefinedProperty->union(self.redefinedProperty.allRedefinedProperties())

[4] This returns the slot corresponding to the supplied property. For redefined properties it will be the slot corresponding to the redefining one.

Note that derived properties will only have slots if they are redefined by a non-derived one so the result may be null.
ClassInstance::propertySlot(Property.p): Slot
result = self.slot->any(definingFeature =
p.applicableDefinition(self.classifier))

[5] This returns the property that defines the slot that will carry the data for the requesting property in an instance of the class c.
Property::applicableDefinition(Class c): Property
applicableDefinition = c.allSlottableProperties().any(p |
p=self or p.allRedefinedProperties()->includes(self))

[6] This returns the single Property with a slot that represents the current owner of the Object based on current instance values; may be null for top level objects.
Object::owningProperty(): Property modeled as ClassInstance::owningProperty(): Property
result = self.classifier.allSlottableProperties()->any(p |
p.opposite <> null and p.opposite.isComposite and self.get(p)<> null)

[7] All the non-redefined properties of an object.
Object::allProperties(): Set(Property) modeled as ClassInstance:: allProperties (): Set(Property)
result = self.classifier.allProperties()

[8] This returns the Properties that subset a derived union
Object::unionedProperties(p: Property): Set(Property)
pre: p.isDerivedUnion

post:
result = self.allProperties->select(sp| sp.subsettedProperty includes p)

[9] This returns all the Constraints for a classifier
CMOF::Classifier::allConstraints(): Set(Constraint)
post:
result = extent().objectsOfType(Constraint)->select(c | c.context = self)

[10] This sets a property value regardless of whether read only (used for initialization)
ObjectInstance::internalSet(Property p, ElementInstance v)
post: (self.propertySlot(p) <> null and self.propertySlot(p).value = v) or
(p.namespace (self.p.namespace.isOclType(Association) and
not (self.allParents() includes p.namespace) and -- allow access to own assoc ends
setLink(p, v)) or
(p.isDerived and result = self.extInvoke('set', p, v))

[11] This unsets a property value regardless of whether read only (used for initialization)
ObjectInstance::internalUnset(Property p)
post: (self.propertySlot(p) <> null and self.propertySlot(p).value = null) or
(p.isDerived and result = self.extInvoke('unset', p, v))

[12] This adds an object to an Extent - only on creation
ExtentImpl::addObject(ObjectInstance o, String suppliedId [0..1]): String
pre: not(self.entry.identifier includes suppliedId)
post: oclIsNew(e) and oclType(e) = IdentifierEntry and
e.object = o and
self.entry includes e
self.entry->select(ex | ex.identifier = e.identifier)->size() = 1 -- the new id is unique and
(suppliedId <> null implies e.identifier = suppliedId)

[13] This removes an object from an extent - only on destruction
ExtentImpl::removeObject(ObjectInstance o)
pre: self.objects includes o
post: let e = self@pre.entry->select(ex|ex.object = o) and
destroyed(e) and
not(self.entry includes e)

[14] This navigates an association end from an object
ObjectInstance::navigate(Property p): Set(ObjectInstance)
pre: p.namespace.isOclType(Association)
post:
-- Find the relevant Links by querying the Extent
-- Issue - need to deal with subsets/unions
let values= extent().objectsOfType(p.namespace)->select(link| link.get(p.opposite) = self)->get(p) and
(p.isUnique implies result = oclAsSet(values))
and not(p.isUnique implies result = values)

[15] Sets an association end

ObjectInstance::setLink(Property p, Element v)

-- If the property is multivalued then break it into individual elements and create links

-- In either case delete existing links

pre: p.namespace.isOclType(Association)

post: let oldValues= extent@pre().objectsOfType(p.namespace)->select(link| link.get(p.opposite) = self)->get(p) and
values->forAll(v| v.delete()) and
(p.upper = 1 implies self.createLink(p, v)) and
(p.upper > 1 implies v->forAll(o| createLink(p, o))

[16] creates an individual link

ObjectInstance::createLink(Property p, ObjectInstance v)

-- Use normal object creation. Assume the existence of the Factory

post: factory.create(p.namespace, Set{ Tuple{p, v}, Tuple{p.opposite, self}})

Annex A
XMI for MOF 2 Core

(normative)

The Package::URI for EMOF is:

http://www.omg.org/spec/MOF/20131001/emof.xmi

And for CMOF is:

http://www.omg.org/spec/MOF/20131001/cmof.xmi

MOF 2 Core directly reuses UML for metamodel representation.

Hence, though EMOF and CMOF have their own package URI, they do not define their own XMI namespace or prefix (which override the Package URI via the org.omg.xmi.nsURI tag).

MOF 2.5 uses UML 2.5 and so:

The namespace uri is: http://www.omg.org/spec/UML/20131001

The namespace prefix is: uml

The exception to the above is the MOF Extension package, for MOF Tags, which is not covered by UML and has namespace as follows:

The namespace uri is: http://www.omg.org/spec/MOF/20131001

The namespace prefix is: mofext

Note: for transition purposes it is possible to transform in both directions between MOF 2 EMOF and CMOF metamodels and compliant metamodels represented in UML 2.5 without loss.

- tag contentType set to “any” for Element

Annex B
Metamodel Constraints in OCL

(informative)

The constraints to be applied to a UML model to validate whether it is a valid metamodel are provided in accompanying files as referenced below. They are executable in the Eclipse OCL environment.

The constraints for EMOF are in: http://www.omg.org/spec/MOF/20131001/CMOFConstraints.ocl

The constraints for CMOF are in: http://www,omg.org.spec/MOF/20131001/EMOFConstraints.ocl

Annex C
Migration from MOF 1.4

(normative)

C.1 General

While MOF 2 is the actual generation of MOF [MOF2], it does not replace or supersede MOF 1.4 [MOF1]. Therefore a normative migration for existing MOF 1.4 metamodels to MOF 2 CMOF metamodels is provided by this annex. MOF 1.4 metamodels can be translated to MOF 2 metamodels based on a straightforward mapping that can be fully automated as described below. Note that attributes that have an obvious direct mapping (e.g., MOF 1.4 ModelElement::name to MOF 2 NamedElement::name) are not listed here.

This Annex addresses the migration of existing MOF 1.4 metamodels to MOF 2 Complete. MOF 1.4 metamodels can be translated to MOF 2 models based on a straightforward mapping that can be fully automated as described below. Note that attributes that have an obvious direct mapping (e.g., MOF 1.4 ModelElement::name to MOF 2 NamedElement::name) are not listed here.

C.2 Metamodel Migration

A valid MOF 1.4 metamodel can be translated to a MOF 2 model. Translation is based on a straightforward mapping that can be fully automated.

MOF 1.4	MOF 2 Mapping
ModelElement::annotation	New instance of Comments::Comment class linked to a corresponding Element via annotatedElement attribute and Comment::body attribute set to the value of the annotation and Comment::usage attribute set to “documentation.”
ModelElement::container	NamedElement::namespace (Note that this is abstract, so appropriate specializations must be used.)
ModelElement::constraints	The association between constraint and constrained element is navigable only from constraint to constrained element, not vice versa. To constrain an element, the element needs to be added to Constraint::context attribute of a given constraint.
Namespace::contents	Namespace::ownedMember (Note that this is abstract, so appropriate specializations must be used.)
GeneralizableElement::isLeaf	Not supported. This attribute can be ignored without losing any information as the attribute constrains the model and not the objects being modeled.
ModelElement::container	NamedElement::namespace (Note that this is abstract, so appropriate specializations must be used.)
ModelElement::constraints	The association between constraint and constrained element is navigable only from constraint to constrained element, not vice versa. To constrain an element, the element needs to be added to Constraint::context attribute of a given constraint.
Namespace::contents	Namespace::ownedMember (Note that this is abstract, so appropriate specializations must be used.)

GeneralizableElement::isLeaf	Not supported. This attribute can be ignored without losing any information as the attribute constrains the model and not the objects being modeled.
GeneralizableElement::isRoot	Not supported. This attribute can be ignored without losing any information as the attribute constrains the model and not the objects being modeled.
GeneralizableElement::supertypes	Classifier::general
Class::isSingleton	MOF 1.4 Class with isSingleton = true is no longer directly supported. It can be simulated by inserting a Constraint on the class: self.metaobject.allInstances.size = 1.
CollectionType	Not directly supported. Can be substituted by an instance of DataType class with one attribute (called 'value') of the same type and multiplicity as the CollectionType.
EnumerationType	Mapped to Enumeration, where strings in the value of the MOF 1.4 EnumerationType::labels attribute are mapped to instances of EnumerationLiteral class with the label string as the name and the same ordering. The enumeration literals are linked to an enumeration via the Enumeration::ownedLiteral attribute.
AliasType	Not supported. Map to a subtype of the concrete data type corresponding to the type that the AliasType points to. Attach any constraints from the AliasType to the subtype.
StructureType	Maps to DataType.
StructureField	Maps to Property owned by a DataType corresponding to the parent StructureType.
Feature::scope	Not supported – all the features in MOF 2 are instance-level.
StructuralFeature::isChangeable	Maps to Property::isReadOnly which has the opposite meaning (i.e., isReadOnly = not isChangeable).
Reference	Redundant as a separate element from the referenced AssociationEnd (which is now a Property). The fact that the AssociationEnd had a Reference means that the new Property corresponding to the AssociationEnd should be owned by the Class not the Association.
Reference::referencedEnd	The Property representing the association itself. Redundant as above.
Reference::exposedEnd	Property::opposite. Redundant as above.
Operation::exceptions	Operation::raisedException.
Exception	An Exception can be any desired subclass of Classifier. By default it should be a Class with the same name as the Exception.
AssociationEnd	Property associated with an Association via memberEnd attribute. To make the Property (i.e., the AssociationEnd) exposed in a class (similarly to using a reference in MOF 1.4), it needs to be owned by the class (while still being a memberEnd of the association).
AssociationEnd::isNavigable	Not supported as defined in MOF 1.4. In MOF 2 all the ends are navigable in terms of MOF 1.4 navigability. MOF 2 defines navigability in terms of being able to navigate to the end directly from a type – this is analogous to having a reference in MOF 1.4.
AssociationEnd::aggregation	Property::isComposite (composite maps to true, none maps to false, shared was underspecified in MOF 1.4 –maps to false).
AssociationEnd::isChangeable	See mapping of StructuralFeature::isChangeable.

Import	PackageImport, PackageMerge or ElementImport classes. If the imported elements are whole packages, then Import maps to PackageImport in case of isClustered=false and PackageMerge in case of isClustered=true (this may need to be revisted in conjunction with the MOF 2 Facility RFP. Note: For importing individual elements inside packages Import maps to ElementImport. Note that in MOF 2, an import makes names within imported packages visible without qualification.
Import::importedNamespace	PackageImport::importedPackage, PackageMerge::mergedPackage, ElementImport::importedElement Note that in MOF 2, an import makes names within imported packages visible without qualification.
Tag	Extension::Tag

C.3 API Migration

This sub clause summarizes the API equivalents between the MOF 1.4 Reflective API, the JMI API and the MOF 2 Core API. Class names are in bold and italics.

Note that many of the gaps for MOF 2 are where the previous APIs have operations specific to the language binding (e.g., related to class proxies) or are convenience operations.

MOF 1.4	**JMI**	**MOF 2**
RefBaseObject	RefBaseObject	Object
refMofId	refMofId	Extent::identifier
refMetaObject	refMetaObject	getMetaclass
refImmediatePackage	refImmediatePackage	
refOutermostPackage	refOutermostPackage	
refItself	Java Object.equals	Element::equals
refVerifyConstraints	refVerifyConstraints	
refDelete	RefObject::refDelete	delete
RefObject	RefFeatured	
refValue	refGetValue	get
refSetValue	refSetValue	set
refUnsetValue	Set value to null	unset
refAddValueBefore	Use of live collections	
refAddValueAt	*ditto*	
refModifyValue	*ditto*	
refModifyValueAt	*ditto*	
refRemoveValue	*ditto*	
refRemoveValueAt	*ditto*	
refInvokeOperation	refInvokeOperation	invoke
Test for value = null	Test for value = null	isSet

	RefObject	
refIsInstanceOf	refIsInstanceOf	isInstanceOfType
	refClass (gets proxy)	
refImmediateComposite	refImmediateComposite	container
refOutermostComposite	refOutermostComposite	
RefBaseObject::refDelete	refDelete	delete
RefClass	RefClass	
refCreateInstance	refCreateInstance	Factory::create
refAllObjects (includeSubtypes=true)	refAllOfType	Extent::elementsOfType (includeSubtypes=true)
refAllObjects (includeSubtypes=false)	refAllOfClass	Extent::objectsOfType (includeSubtypes=false)
	refCreateStruct	Factory::createFromString
	refGetEnum	Factory::createFromString
refAssociation	refAssociation	
refAllLinks	refAllLinks	Extent::linksOfType
refLinkExists	refLinkExists	Extent::linkExists
refQuery	refQuery	Extent::linkedElements
refAddLink	refAddLink	Factory::create
refAddLinkBefore	Use live collections	
refModifyLink	*ditto*	
refRemoveLink	refRemoveLink	delete
	refAssociationLink (datatype – tuple)	
RefPackage	RefPackage	Extent
refMofId (inherited)	refMofId (inherited)	
refMetaObject(inherited)	refMetaObject(inherited)	
refImmediatePackage(inherited)	refImmediatePackage(inherited)	
refOutermostPackage(inherited)	refOutermostPackage(inherited)	
refItself(inherited)	Java Object.equals	
refClassRef	refClass	
refAssociation	refAssociation	
refPackage	refPackage	
	refAllPackages	
	refAllAssociations	
	refCreateStruct	
	refGetEnum	
RefBaseObject::refDelete	refDelete	

		useContainment (attribute)
		elements
		identifier
		element

Annex D
Bibliography

(informative)

[CORBA] Common Object Request Broker Architecture (CORBA) Specification http://www.omg.org/spec/CORBA/3.3/
ISO/IEC 19500:2012 Information technology - Object Management Group Common Object Request Broker Architecture (CORBA)

[MOFFOL] MOF Facility Object Lifecycle (MOFFOL) http://www.omg.org/spec/MOFFOL/2.0/

[MOFVD] Meta Object Facility (MOF) Versioning and Development Lifecycle Specification http://www.omg.org/spec/MOFVD/2.0

[MOFM2T] MOF Model to Text Transformation Language http://www.omg.org/spec/MOFM2T/1.0/

[QVT] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification http://www.omg.org/spec/QVT/1.1

[SMOF] MOF Support for Semantic Structures http://www.omg.org/spec/SMOF/1.0/

[UML1] OMG Unified Modeling Language Specification http://www.omg.org/spec/UML/1.4
ISO/IEC 19501:2005 Unified Modeling Language Specification

[UML2Inf] OMG Unified Modeling Language (OMG UML), Infrastructure http://www.omg.org/spec/UML/2.4.1/Infrastructure/
ISO/IEC 19505-1:2012 Information technology - Object Management Group Unified Modeling Language (OMG UML), Infrastructure

[UML2Sup] OMG Unified Modeling Language (OMG UML), Superstructure http://www.omg.org/spec/UML/2.4.1/Superstructure/
ISO/IEC 19505-1:2012 Information technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure

[XMI2] XML Metadata Interchange http://www.omg.org/spec/XMI/2.0
ISO/IEC 19503:2005 XML Metadata Interchange Specification

[XMI24] XML Metadata Interchange http://www.omg.org/spec/XMI/2.4.2
ISO/IEC 19509:2014 Information technology - Object Management Group XML Metadata Interchange (XMI)

[XSD-D] W3C Recommendation XML Schema Part 2: Datatypes Second Edition http://www.w3.org/TR/2004/REC-xmlschema-2-20041028