
Meta Object Facility (MOF) 2.0 Core Specification

This OMG document replaces the submission document (ad/03-04-07) and the Draft Adopted specification (ptc/03-08-06). It is an OMG Final Adopted Specification and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by November 7, 2003.

You may view the pending issues for this specification from the OMG revision issues web page *<http://www.omg.org/issues/>*.

The FTF Recommendation and Report for this specification will be published on April 30, 2004. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

Date: October 2003

Meta Object Facility (MOF) 2.0 Core Specification

version 2.0

Final Adopted Specification

ptc/03-10-04

Copyright © 2003, Adaptive
Copyright © 2003, Ceira Technologies, Inc.
Copyright © 2003, Compuware Corporation
Copyright © 2003, Data Access Technologies, Inc.
Copyright © 2003, DSTC
Copyright © 2003, Gentleware
Copyright © 2003, Hewlett-Packard
Copyright © 2003, International Business Machines
Copyright © 2003, IONA
Copyright © 1997-2003, Object Management Group
Copyright © 2003, MetaMatrix
Copyright © 2003, Softeam
Copyright © 2003, SUN
Copyright © 2003, Telelogic AB
Copyright © 2003, Unisys

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.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBA services™, CORBA facilities™, CORBA med™, CORBA net™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

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.

ISSUE REPORTING

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 & Specifications, Report a Bug/Issue.

Table of Contents

1	Scope	1
2	Conformance	1
3	Normative References	1
4	Terms and Definitions	2
5	Symbols	3
6	Additional Information	3
	Part I - Introduction	5
7	MOF Architecture	7
	7.1 MOF 2.0 Design Rationale	7
	7.2 How many "Meta Layers"?	8
	7.3 Reuse of Common Core Packages by UML 2.0 and MOF 2.0	9
8	Language Formalism	11
	8.1 Metamodel Specification	11
	8.2 Using Packages to Partition and Extend Metamodels	11
	8.3 Changes from MOF 1.4 (optional)	11
	8.4 Null	12
	Part II - Capabilities	13
9	Reflection	15
	9.1 Object	15
	9.2 Factory	18
10	Identifiers	21
	10.1 Extent	21
	10.2 Package	22
	10.3 Property	23
	10.4 URIExtent	23
	10.5 MOF::Common	25
11	Extension	27
	11.1 Tag	27
	Part III - The MOF Model	29
12	The Essential MOF (EMOF) Model	31
	12.1 Introduction	31
	12.2 EMOF Extensions to Basic	33
	12.3 EMOF Merged Model	35
	12.4 Imported Elements from UML 2 Core	35

12.5	Merged Elements from MOF	38
12.6	EMOF Constraints	38
12.7	EMOF Definitions and Usage Guidelines for the Basic Model	39
13	CMOF Reflection.....	41
13.1	Link	42
13.2	Argument	43
13.3	Factory	44
13.4	Extent	45
13.5	Exception	46
14	The Complete MOF (CMOF) Model	47
14.1	CMOF Extensions to Core::Constructs	48
14.2	PackageMerge	49
14.3	Imported Elements from UML 2 Core	51
14.4	Imported Elements from MOF	52
14.5	CMOF Constraints	53
14.6	CMOF Extensions to Capabilities	53
Part IV	- Abstract Semantics	55
15	CMOF Abstract Semantics	57
15.1	Approach	57
15.2	MOF Instances Model	57
15.3	Notes	60
15.4	Object Capabilities.....	60
15.5	Link Capabilities	62
15.6	Factory Capabilities	63
15.7	Extent Capabilities.....	64
15.8	Additional Operations	66
16	Migration From MOF 1.4	71
16.1	Metamodel Migration	71
16.2	API Migration	73
Part V	- Appendices	77
A.	XSD and XMI for MOF 2.0	79
B.	JMI Java Language Mapping	81

1 Scope

This MOF 2.0 Core specification is in response to the Object Management Group Request For Proposals ad/01-11-14 (*MOF 2.0 Core RFP*). This MOF 2.0 specification is based on the progress made in the following OMG Specifications and OMG work in progress:

- MOF 1.4 Specification -- MOF 2.0 is a major revision of the MOF 1.4 Specification. MOF 2.0 addresses issues deferred to MOF 2.0 by the MOF 1.4 RTF. For information on migration from MOF 1.4 to MOF 2.0 please refer to the “Migration from MOF 1.4” chapter.
- UML 2.0 Infrastructure: ptc/03-09-15 -- MOF 2.0 reuses a subset of the UML 2.0 Infrastructure Library packages.
- MOF 2.0 XMI submission: ad/03-04-04 which is an update to the XMI 2.0 specification to address MOF 2.0 and UML 2.0 requirements.

2 Conformance

There are two compliance points:

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

Compliant products shall support one or more of the following technology mappings: MOF 2.0 XMI (ad/2003-04-04), MOF 2.0 IDL, and MOF 2.0 JMI. Additional conformance rules based on MOF 2.0 IDL and MOF 2.0 JMI will be specified in separate specifications.

3 Normative References

Readers of this MOF 2.0 specification are expected to be familiar with the UML Infrastructure V 2.0 specification. The MOF 2.0 model packages Essential MOF (EMOF) and Complete MOF (CMOF) are constructed by importing the UML2 Infrastructure Library packages.

The main body of the document describes the technical specification itself. This specification is made up of the following documents:

- MOF 2.0 Core Specification (ptc/03-08-06)
- MOF 2.0 XSD, XMI, and MDL files (ad/2003-04-08)

4 Terms and Definitions

Editorial Comment: The FTF needs to review and complete this section -- or delete if applicable.

5 Symbols

Editorial Comment: The FTF needs to review and complete this section -- or delete if applicable.

6 Additional Information

6.1 Mailing Lists

The following mailing list is available for requesting information and reporting issues:

- issues@omg.org — to handle issue management and revision to the specifications, or
- via the OMG web site.

6.2 Technical specification

The technical specification is presented in the main body of this document. It comprises a set of new packages that will result in a major revision to the architecture and foundation constructs of the MOF 1.4 specification.

6.3 How to Read this Specification

Part one is best read first, beginning to end. The rest is organized based on package structure and can be read in any order.

6.4 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Adaptive
- Borland
- Ceira Technologies
- Compuware
- Data Access Technologies
- DSTC
- Gentleware Intellicorp
- Hewlett-Packard
- Hyperion
- International Business Machines
- IONA
- Kinetium
- MetaMatrix

- Project Technology
- SOFTEAM
- Sun Microsystems
- Telelogic AB
- Unisys
- University of Kent
- University of York
- X-Change Technologies Group
- 88solutions

The submitters acknowledge the generous help in preparation of this specification from others outside the team. In particular, the U2P, UU and 3C teams proposing UML 2 have contributed much to this effort.

Part I - Introduction

The Internet has connected data sources and applications across the planet driving an expectation of easy and transparent data exchange between applications. Incompatible metadata across different systems have been a primary limitation on data exchange. Metadata are data about data. They are the data used by tools, databases, middleware, etc. to describe structure and meaning of data.

Unfortunately, many applications use proprietary models of metadata. The differences between metadata models impede data exchange across application boundaries. Resolving these differences will accelerate integration of important applications that require interchange, such as data warehousing, business intelligence, e-commerce, and information portals.

The Meta Object Facility (MOF), an adopted OMG standard, (latest revision MOF 1.4) provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems. Examples of these systems that use MOF include modeling and development tools, data warehouse systems, metadata repositories etc. A number of technologies standardized by OMG, including UML, MOF, CWM, SPEM, XMI, and various UML profiles, use MOF and MOF derived technologies (specifically XMI and more recently JMI which are mappings of MOF to XML and Java respectively) for metadata-driven interchange and metadata manipulation.

MOF has contributed significantly to some of the core principles of the emerging 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 standard 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-IDL mapping in the MOF specification, MOF-to-XML DTD mapping in the XMI specification, MOF-to-XML Schema mapping in the XMI production of XML Schema specification, and MOF-to-Java in the JMI spec).

The OMG adopted the MOF 1.1 specification in November 1997 coincident with the adoption of UML 1.1 as a result of unprecedented collaboration between the vendors proposing UML 1.1 and MOF 1.1 specifications. This collaboration continues to this day as the vendors working on UML 2.0 Infrastructure and MOF 2.0 are attempting even greater reuse (as required by the OMG RFPs) and integration of modeling concepts to provide a solid foundation for MDA. It is fair to mention that this integration work has proven to be more challenging than expected but is expected to drive the next generation of model and metadata driven systems in the industry.

Since then, MOF Revision Task Forces have produced several minor revisions, the most recent being the MOF 1.4 specification, which was adopted in October 2001.

The use of MOF and XMI over the last few years has raised numerous application and implementation issues by vendors. As of the time of this writing over 150 formal usage and implementation issues have been submitted to the OMG for consideration. While a vast majority of these issues have been addressed by MOF 1.4, some are considered too major to be addressed by a Revision Task Force (RTF), and therefore, a series of MOF 2.0 RFPs (six so far: MOF 2.0 Core, MOF 2.0 IDL Mapping, MOF 2.0 XMI Mapping, MOF 2.0 Versioning and MOF 2.0 Query/View/Transformations, MOF 2.0 Facility RFP) have been issued. One more (MOF 2.0 Facility RFP) has been presented to the ADTF but not issued.

The reader must keep in mind that the individual RFPs and the proposals can be used incrementally and independently, and this modularity is a design goal of MOF 2.0. For example, vendors can implement the MOF 2.0 Model as a framework for metamodeling and metadata representation and management without using MOF 2.0 IDL or MOF 2.0 Java mapping. This was considered very important to provide more choice for MOF implementors. Likewise, not all MOF vendors agree that metadata versioning and federation are required features of all MOF systems, and hence the decision to separate out more focused requirements in separate RFPs. Time will tell if this component-based approach to building specifications will be successful, but the submitters are convinced that we are on the right track.

This proposed MOF 2.0 specification integrates and reuses the complementary U2P UML 2.0 Infrastructure Proposal to provide a more consistent modeling and metadata framework for OMG's Model Driven Architecture. UML 2.0 provides the modeling framework and notation, MOF 2.0 provides the metadata management framework and metadata services. The manner in which the proposal addresses individual RFP requirements is explained in the Preface of this proposal. Considerable progress has been made in eliminating overlapping modeling constructs in UML 1 and MOF 1 specifications (For example the confusion between MOF References and UML AssociationEnds has been eliminated). More importantly the modeling constructs from the UML2 Infrastructure Library are reused (using import) by both the MOF 2, U2P UML2 Infrastructure and U2P UML2 Superstructure proposals.

One of the challenges the MOF 2.0 Core and MOF 2.0 XMI Mapping submissions face is to maintain a stable interchange model (XMI) while MOF 2 and UML 2 are changing quite significantly. To accomplish this, we have begun applying the design principles that have been used in the XMI for XML Schemas and now the MOF 2.0 XMI mapping submission. This is to use a very small subset of the modeling concepts in MOF (We call this Essential MOF or EMOF which basically models simple classes with attributes and operations) to fix the basic mapping from MOF to XML and Java. Additional mapping rules are provided in a manner consistent with XMI 2.0 for more complex modeling constructs. Please refer to the MOF 2.0 XMI proposal : [ad/2002-12-07](#) for more details.

The next chapters describe the language architecture, proofs of concepts, and formalism approach used to define MOF 2.0.

7 MOF Architecture

This chapter describes the architecture of the MOF and how it serves as the platform-independent metadata management foundation for MDA. The chapter also summarizes major architectural decisions that influenced the design of MOF 2.0. Finally, the relationship of MOF 2.0 to UML 2.0 and the use of MOF to instantiate UML 2.0 and future OMG metamodels is summarized.

7.1 MOF 2.0 Design Rationale

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 2.0 has strongly influenced and, at the same time, has been strongly influenced by the U2P UML 2.0 Infrastructure proposal because of a shared vision of reusing the core modeling concepts between UML 2.0, MOF 2.0 and other emerging OMG metamodels. The fact that some of the same companies and designers worked on both the specs and championed these design principles has made this unification and reuse possible. 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 UML2 Infrastructure library in both the MOF2 and U2P UML2 proposals. 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 U2P UML profiles.
- Broader tool support for metamodeling (any UML modeling tool can be used to model metadata more easily)

While much progress has been made in unification of the modeling concepts in MOF 2 and UML 2, the goal of reuse of meta model packages across object and non object systems continues to be challenging. To address this the initial MOF 2.0 Core proposal included a chapter titled “Common concepts”. In the revised proposal (in part to meet current submission deadlines and because this task proved to be more difficult than anticipated) we have scaled back these goals to focus on reuse between just UML and MOF. The MOF2 submission team recommends additional OMG RFPs be issued to address this broader reuse of metamodel packages between object and non object systems.

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 implementors 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 2.0, MOF 2.0 and additional key metamodels (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. Note that this was also an overriding design goal for the U2P UML 2.0 Infrastructure proposal. 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 2.0 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 2.0 and UML 2.0 Specifications.
4. Ensure that MOF 2.0 is technology platform independent and that it is more practical to map from MOF 2.0 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 2.0. 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 2.0. 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 2.0 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 2.0 models the concept of identity. The lack of this capability in MOF, UML, CWM etc., made interoperability of metadata difficult to implement. The submitters understand that modeling identity is not easy, but we plan to show its usefulness in a simple domain - identity of metadata first. A key design goal is to make it easy to map this model of identity to W3C identity 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.

7.2 How many “Meta Layers”?

One of the sources of confusion in the OMG suite of standards is the perceived rigidity of a ‘Four layered metamodel architecture’ which 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 1.4 Reflection interfaces allow traversal across any number of metalayers recursively. Note that most systems use a small

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 2.0 Infrastructure, UML 1.4 and MOF 1.4 specification - MOF/UML/User Model/User Object). MOF 1 and MOF 2.0 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 2.0 with its reflection model can be used with as few as 2 levels and as many levels as users define.

7.3 Reuse of Common Core Packages by UML 2.0 and MOF 2.0

The U2P UML 2 Infrastructure Draft proposal uses fine grained packages to bootstrap the rest of UML 2.0. A design goal (and RFP Requirement) is to reuse this infrastructure in the definition of the MOF 2.0 Model. In MOF 2.0 this reuse is simply accomplished by using a standard CMOF extensibility mechanism -- importing, combining, and merging existing MOF 2.0 compliant packages. Importing packages makes model elements contained in the imported package visible in the importing package. Combining packages are used to define a new package that consists of the model elements of the combined and combining packages. Merging packages extends model elements in the merged package with new feature deltas from the merging package. The details are covered in Chapter 10, "CMOF". Note that we have designed both the U2P UML 2.0 model and the MOF 2.0 model to be compliant to MOF 2.0 and be instantiable from MOF 2.0. See Chapter 2, "Proof Of Concept" for more details.

Standard class modeling concepts (importing, subclassing, adding new classes, associations and adding associations between existing classes) are used for MOF 2.0 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 2.0 as well as in any other MOF 2.0 compliant model.

Figure 1 shows that MOF imports the UML Core, and then extends the Core with additional packages.

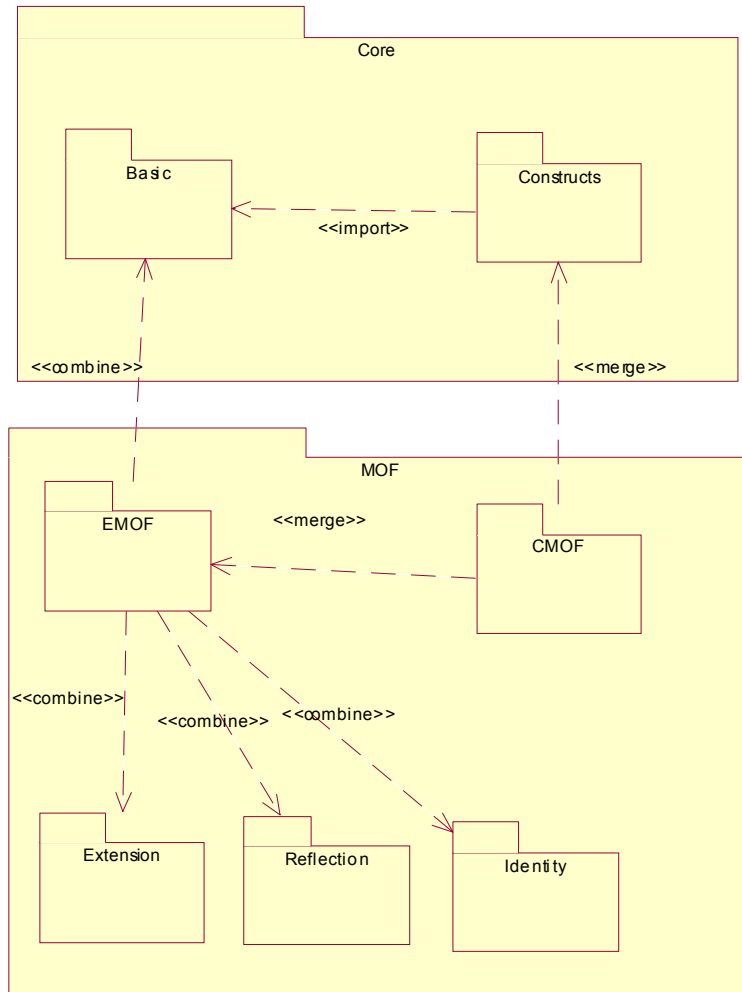


Figure 1 - MOF imports from the UML Core

8 Language Formalism

This chapter explains techniques used to describe the MOF. The MOF is described using both textual and graphic presentations. The specification 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 MOF2 specification reuses much of the formalisms in the U2P Infrastructure proposal. 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, combine, and merge semantics from its CMOF description. As a result, EMOF and CMOF are described using themselves, and each is derived from, or reuses part of the U2P Infrastructure.

8.1 Metamodel Specification

Please refer to the chapter “Language Formalism” in the U2P UML Infrastructure Specification. The CMOF model reuses the same formalisms and in fact simply imports, and merges packages in the U2P UML2 Infrastructure Library. EMOF is somewhat simpler as described in the next section.

8.2 Using Packages to Partition and Extend Metamodels

Packages in UML2 Constructs 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 adding new metamodeling features through extension. Package merging allows the merging package to specialize model elements in the merged package that have matching names. Classes in the merging package specialize similarly named classes in the merged package adding new features. Redefinitions are used in the merging package to override features in the merged superclasses to provide stronger types in the merging package.

Package merge does not define a new, separate package, but rather maintains a dependency between the merged and merging packages. Effectively, classes in the merging package subclass matching classes in the merged classes and the packageMerge is converted into a packageImport allowing model elements that don't match to be visible in the merging package. However, this form of package merge does not always produce the desired result because it introduces classes with the same name in different packages, extra superclasses, and redefinitions, which add complexity to the result. EMOF does not support redefinitions, so this form of package extension cannot be used to produce EMOF. To address this problem, CMOF extends Constructs::PackageMerge with an additional attribute that determines if the merge is extending a package as described in the U2P Infrastructure, or defining a new package. This new kind of packageMerge is denoted using the stereotype <<combine>> instead of <<merge>> on the dependence between the merging source and merged target packages. In the combined form of package merge, all the model elements of the merged package are copied into the merging package. Model elements that match are merged into a single model element instead of using inheritance. The dependency between the packages is removed. The merging package stands alone and has all the features of both packages. See Chapter 10, “CMOF” for detailed semantics.

8.3 Changes from MOF 1.4 (optional)

Here, changes compared with MOF 1.4 are described as appropriate. See also Chapter 7 “Migration from MOF 1.4 to MOF 2.0”.

8.4 Null

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

Part II - Capabilities

The MOF models use a number of building-block concepts meant for widespread reuse by other models and metamodels. These separate concerns or “capabilities” contains several packages addressing different modeling and metadata management concerns. These packages are then merged into EMOF and CMOF, and other metamodels as needed in order to extend those models with the capabilities. The packages described in this part are:

- **Reflection:** extends a model with the ability to be self describing
- **Identity:** 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 various packages making up the supported capabilities are instances of CMOF::Package, and all of its contents are instances of classes in the CMOF Model.

The following chapters describe each of the packages making up the supported capabilities.

9 Reflection

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

Note – The Factory class has some overlap with the MOF Life cycle RFP and will need to be reconciled when submissions for that RFP are adopted.

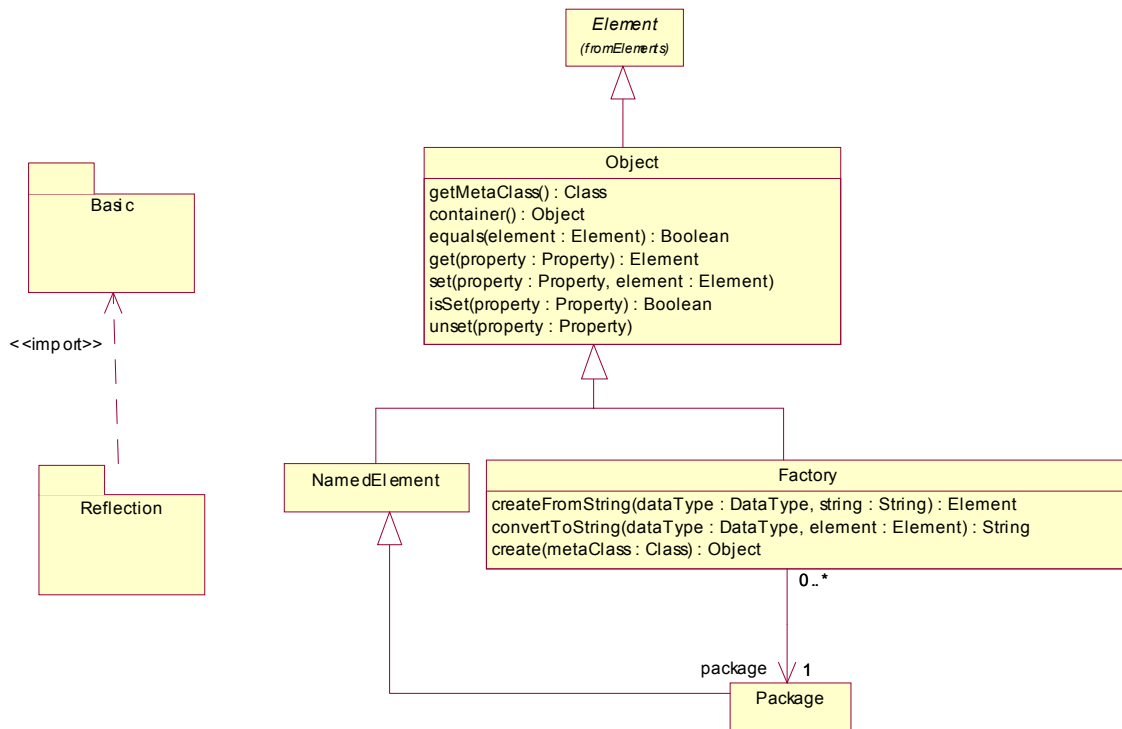


Figure 2 - The Reflection package

9.1 Object

Every Object has a Class which describes its properties and operations. The Object is an Instance of this Class. Object extends Abstraction::Elements::Element. All model elements that specialize Reflection::Object inherit reflective capabilities. In particular, this includes all model elements from UML2 Infrastructure.

All the conditions in “EMOF Definitions and Usage Guidelines for the Basic Model” on page 39 are also part of the reflective behavior

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

Properties

No additional properties.

Operations

getMetaClass() : Class

Returns the Class that describes this object.

container(): Object

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

equals(element: Element): Boolean

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

get(property: Property) : Element

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 ReflectiveSequence containing the values of the Property. If there are no values, the ReflectiveSequence returned is empty. ReflectiveSequence is defined in section “MOF::Common” on page 24.

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

set(property: Property, element: Element)

If the Property has multiplicity upper bound = 1, set() atomically updates the value of the Property to the Element parameter. If Property has multiplicity upper bound >1, the Element may be either a ReflectiveCollection or a ReflectiveSequence. The behavior is identical to the following operations performed atomically:

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

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(element)` returns false and Property has multiplicity upper bound = 1

Exception: throws `ClassCastException` if Element is not a ReflectiveSequence 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 elements in the list is > 0.

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

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 `ReflectiveSequence` of values of the Property. The behavior is identical to the following operations performed atomically:

```
ReflectiveSequence 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()`.

Constraints

No additional constraints.

Semantics

Class `Object` is the superclass of all model elements in MOF, and is the superclass of all instances of MOF model elements. Each object can access its `metaClass` in order to obtain a `Class` that provides a reflective description of that object. By having both MOF and instances of MOF be rooted in class `Object`, MOF supports any number of meta layers as described in Chapter 1, “MOF Architecture”.

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 object is created. `isSet=false`
- If the value of that property is later explicitly set, even to the default value, `isSet=true`
- 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 object 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. In the worst case it can be implemented by having an additional boolean, but usually for a particular implementation it can be implemented more efficiently (e.g. by having an internal distinguished value used to represent "no value set"). 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

Object 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.

Changes from MOF 1.4

To be added.

9.2 Factory

Note – This section will need to be reconciled with the work underway in MOF lifecycle RFP.

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): Element

Creates an Element 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, element: Element): String

Creates a String representation of the Element. 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()`.

create(metaClass: Class): Object

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

All properties of the object 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 object'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: $0 \leq \text{LowerBound} \leq \text{UpperBound}$ required
- [5] Property: $1 \leq \text{UpperBound}$ required
- [6] Enforcement of read-only properties is optional in EMOF
- [7] Properties of type `Class` must be unique
- [8] Properties of type `Class` cannot have defaults
- [9] Multivalued properties cannot have defaults
- [10] Property: Container end must not have `upperBound > 1`, a property can only be contained in one container.
- [11] Property: Only one end may be composite
- [12] Property: Bidirectional opposite ends must reference each other
- [13] Property and `DataType`: Default value must match type

Items 3-13 apply to all Properties of the Class.

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

Semantics

Rationale

Changes from MOF 1.4

None.

10 Identifiers

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

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.

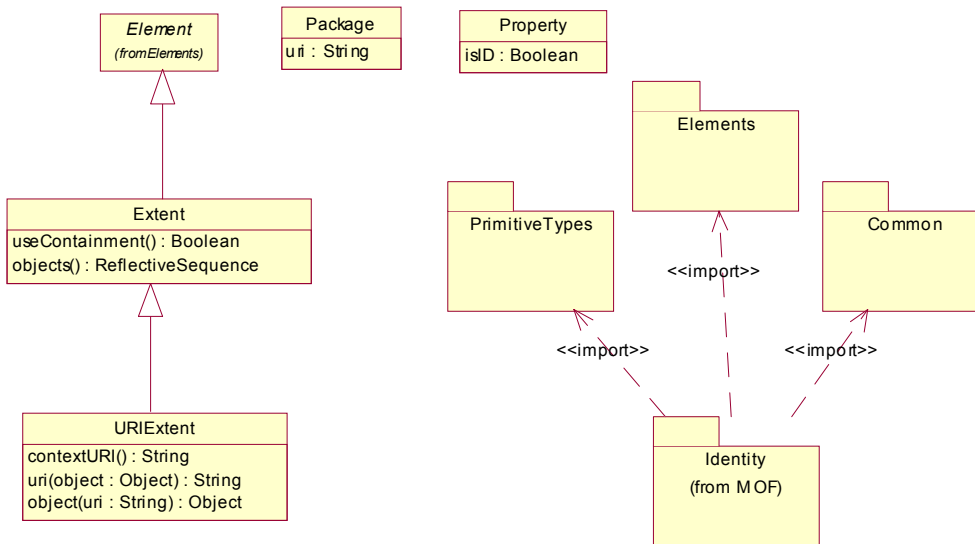


Figure 3 - The Identities package

10.1 Extent

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

Properties

No additional properties.

Operations

useContainment(): Boolean

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

objects(): ReflectiveSequence

Returns a ReflectiveSequence of the objects directly referenced by this extent. If exclusive()==true, these objects must have container()==null. Extent.objects() is a reflective operation, not a reference between Extent and Object. See Chapter 4, “Reflection” for a definition of ReflectiveSequence

Constraints

No additional constraints.

Semantics

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

Rationale

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

Changes from MOF 1.4

MOF 1.4 defined that objects have identity based on a MOF id, but did not provide a way to model identity criteria. In MOF 1.4, MOF id could be queried, but not controlled.

MOF 1.4 assumes creation-based identity for objects of classes. MOF 2 adds explicit specification of creation-based identity MOF2 also reuses the URI identity mechanism for better integration with current XML standards.

10.2 Package

Identity extends Basic::Package with a URI that can be used as an external identifier for a package.

Properties

- **url:** String Provides an identifier for the package that can be used for many purposes. A URI is the universally unique identification of the package following the IETF URI specification, RFC 2396 <http://www.ietf.org/rfc/rfc2396.txt>. UML 1.4 and MOF 1.4 were assigned URIs to their outermost package. The package URI appears in XMI files when instances of the package's classes are serialized.

Operations

No additional operations.

Constraints

No additional constraints.

Semantics

.The URI assigned to the package should be unique and remain unchanged.

Rationale

URIs are the universal standard for identification of all information accessible by computer software.

Changes from MOF 1.4

None.

10.3 Property

Identity extends Basic::Property with the ability to designate a property as an identifier for the containing object.

Properties

- isID: Boolean [0..1] True indicates this property can be used to uniquely identify an instance of the containing Class. Only one Property in a class may have isID==true.

Operations

No additional operations.

Constraints

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

Semantics

A Property with isID==true may be used as part of the URI identifying an object instance.

Rationale

Objects must have identity. The Property isID formalizes this capability in the metadata describing the object..

Changes from MOF 1.4

None.

10.4 URIExtent

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

Properties

No additional properties.

Operations

contextURI(): String

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

uri(object: Object): String

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

object(uri: String): Object

Returns the Object identified by the given URI in the extent. Returns Null if there is no object in the extent with the given URI. Note the Object does not (necessarily) contain a property corresponding to the URI. The URI identifies the object in the context of the extent. The same object 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 identity properties (isID==true).

Rationale

URIs are the defacto standard identity mechanism for the Web and are therefore useful for identifying MOF objects and navigating links between them.

Changes from MOF 1.4

URIExtent did not exist in MOF 1.4.

10.5 MOF::Common

This package defines common elements used throughout MOF.

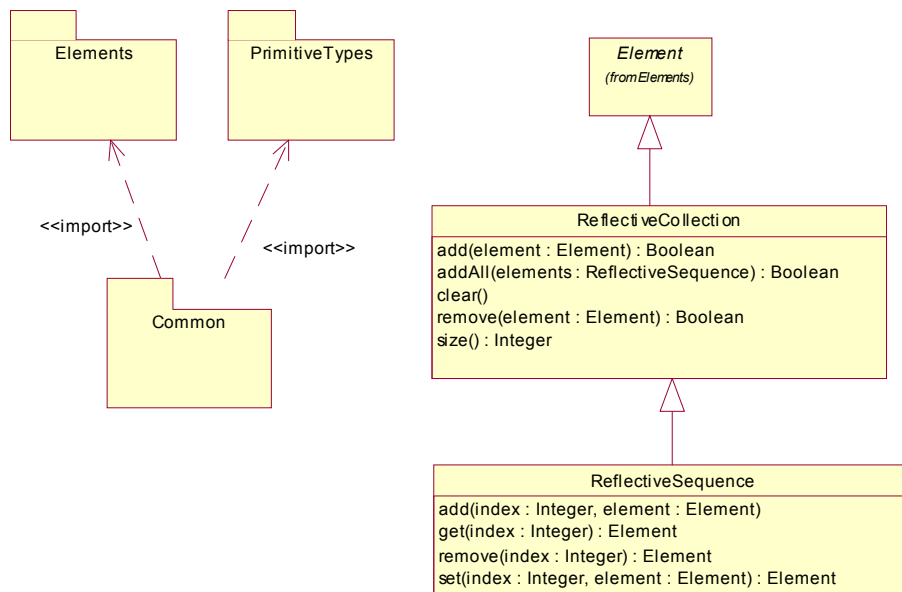


Figure 4 - The Common package.

10.5.1 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(element: Element): Boolean

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

addAll(elements: ReflectiveSequence): Boolean

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

clear()

Removes all elements from the collection.

remove(element: Element): Element

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

size(): Integer

Returns the number of elements in the collection.

10.5.2 ReflectiveSequence

ReflectiveSequence is a subclass of ReflectiveCollection that is used for accessing ordered properties with more than one possible value. Modifications made to ReflectiveSequence update the Object's values for that property atomically. Modifications made to the ReflectiveSequence update the Object'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 $0 \leq \text{index} < \text{size}()$ 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, element: Element)

Adds element to the specified index in the sequence, shifting later elements.

get(index: Integer): Element

Returns the element at the given index in the sequence.

remove(index: Integer): Element

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

set(index: Integer, element: Element): Element

Replaces the element at the specified index with the new element. The removed element is returned.

Behavior of particular operations defined in ReflectiveCollection is the following when applied to a ReflectiveSequence:

add(element: Element): Boolean

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

addAll(elements: ReflectiveSequence): Boolean

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

remove(element: Element): Boolean

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

11 Extension

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.

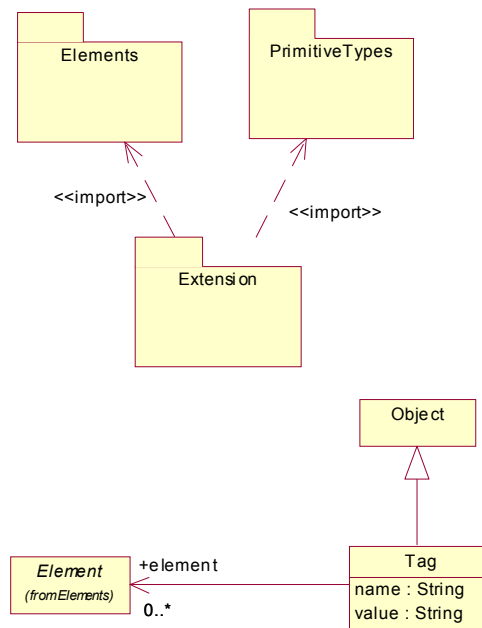


Figure 5 - The Identities package.

11.1 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.

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.

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 provide simple, dynamic extensions.

Changes from MOF 1.4

Part III - The MOF 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).

Metamodels provide a platform independent mechanism to specify the following:

- The shared structure, syntax and semantics of technology and tool frameworks as metamodels
- A shared programming model for any resultant metadata (using Java, IDL etc)
- A shared interchange format (using XML)

The shared programming model and interchange formats can be generated using standardized mappings such as XMI and JMI which are transformations (mappings) of metamodels to specific technologies/languages etc. The use of metamodels in combination with reflection enables generative modeling and programming,

The MOF 2 Model builds on a subset of UML2Infrastructure which provides concepts and graphical notation for the MOF Model. The MOF 2 Model is made of two main packages, Essential MOF (EMOF) and Complete MOF (CMOF) which are described in subsequent chapters. The MOF Model also includes additional capabilities defined in separate packages including support for, identity, additional primitive types, reflection, and simple extensibility through name-value pairs.

To simplify the MOF so as to facilitate ease of implementation and conformance while maximizing interoperability and convenient interchange, MOF 2 defines EMOF as a kernel metamodeling capability defined as a limited conformance level. For the purposes of this specification, the MOF 2 model refers to Complete MOF, the CMOF model. References to EMOF are always explicit.

The following chapters describe the:

- Model Packages Essential MOF (EMOF) and Complete MOF (CMOF),
- MOF Model's requirements and use of the UML 2 Infrastructure, and migration from MOF1.4 to MOF2.

12 The Essential MOF (EMOF) Model

12.1 Introduction

This chapter defines Essential MOF, which is the subset of MOF that closely corresponds to the facilities found in OOPs 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 chapter) reuse the UML2 Infrastructure. The motivation behind this goal is to lower the barrier to entry for model driven tool development and tool integration.

The EMOF Model merges (merge type define) the Basic and Abstractions package from UML2 and defines a few extensions. EMOF does not add any new classes. The EMOF model merges the Reflection, PrimitiveTypes, Identity, and Extension packages to provide services for discovering, manipulating, identifying, and extending metadata.

EMOF, like all metamodels in the MOF 2 and UML 2 family, is described as a CMOF 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 CMOF model. This chapter provides the CMOF model of EMOF, and the complete, merged EMOF model. Note that EMOF uses MergeKind::define to specify the 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 “MOF Abstract Semantics” on page 57 are optional for EMOF.

The relationship between EMOF and InfrastructureLibrary::Core::Basic requires further explanation. EMOF combines Basic with the MOF capabilities and a few extensions of its own that are described below. Ideally, EMOF would just extend Basic 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 between Basic::Element and Basic::NamedElement, which requires the merge. As a result of using MergeKind::define, EMOF is a separate model that combines Basic, but does not inherit from it. See Chapter 9 “CMOF” for details on different merge semantics. As a result, an instance of a Basic model is not an instance of an EMOF model, even though the classes all have the same names, properties, and operations. EMOF could have used MergeKind::extend to extend the Basic model with subclasses, but then either redefinitions are required to tighten the association types in the corresponding EMOF subclasses, or applications using EMOF would have to downcast the Basic types to EMOF types everywhere they are used. In addition, the EMOF model would be complicated by multiple packages containing classes having the same names, and lots of unfortunate subclasses.

To avoid these issues, EMOF uses MergeKind::define to create a new package from merging Basic and EMOF instead of extending and inheriting from Basic. Note however that even though Basic models are not EMOF models, EMOF is directly compatible with Basic XMI files. Defining EMOF as a package merge also ensures EMOF will get updated with and changes to Basic. The reason for specifying the complete, merged EMOF model in this chapter 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.

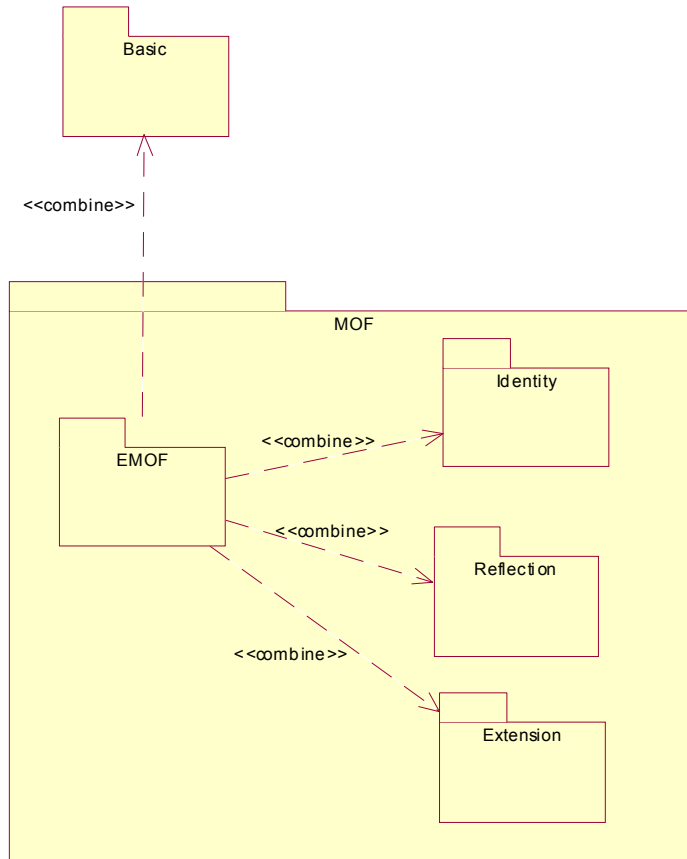


Figure 6 - EMOF Model- Overview

The EMOF model provides the minimal set of elements required to model object-oriented systems. The next section describes the EMOF extensions to Basic.

12.2 EMOF Extensions to Basic

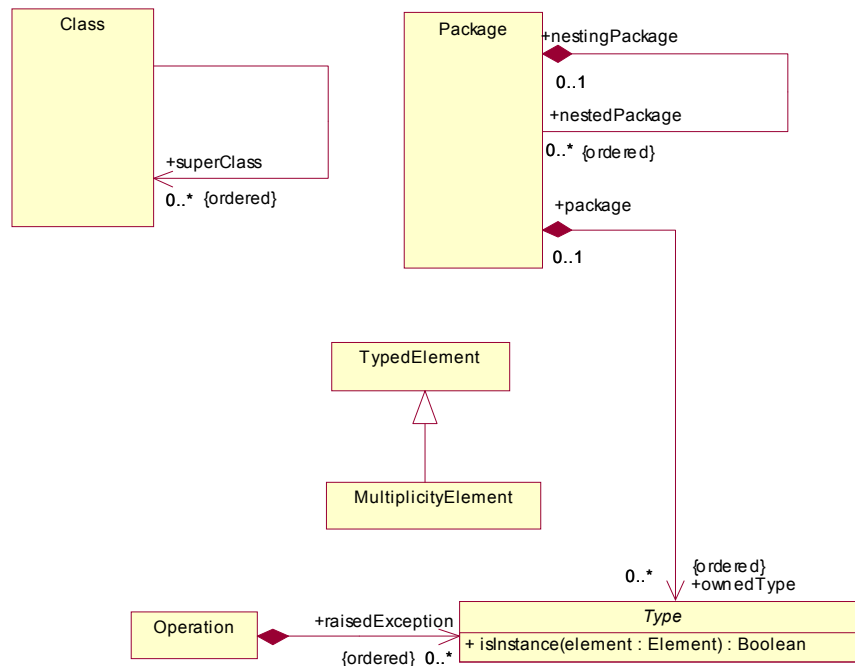


Figure 7 - EMOF Extensions

12.2.1 MultiplicityElement

MultiplicityElement specializes TypedElement in Basic in order to eliminate the need for multiple inheritance for those typed elements that have multiplicity (Property, Operation, Parameter).

Properties

No additional properties.

Operations

No additional operations.

Constraints

No additional constraints.

Semantics

Rationale

Eliminating multiple inheritance simplifies the EMOF model.

Changes from MOF 1.4

None.

12.2.2 Operation

Properties

No additional properties.

Operations

No additional operations.

Constraints

[1] `raisedExceptions` are ordered in EMOF.

Semantics

.

Rationale

Changes from MOF 1.4

None.

12.2.3 Type

Type is included in EMOF to add the ordered constraint to `Operation::raisedException`.

Properties

No additional properties.

Operations

`isInstance(element : Element) : Boolean`

Returns true if the element is an instance of this type or a subclass of this type. Returns false if element is null.

Constraints

No additional constraints.

Semantics

If a class, its properties, or its operations, are modified, the behavior of existing instances is implementation-specific

Rationale

Changes from MOF 1.4

None.

12.2.4 TypedElement

TypedElement is included in EMOF in order to provide a supertype for MultiplicityElement.

Properties

No additional properties.

Operations

No additional operations.

Constraints

No additional constraints.

Semantics

.

Rationale

12.3 EMOF Merged Model

This section provides the complete EMOF model merged with Basic and the MOF capabilities. It is completely specified in EMOF itself after applying the package merge semantics described in Chapter 9, “CMOF”. The description of the model elements, other than the extensions specified above, is identical to that found in UML 2 Infrastructure and is not repeated here.

12.4 Imported Elements from UML 2 Core

By merging the Basic package from the UML 2 Core, the EMOF Model’s members include the following packages from Abstractions as well as Basic, as listed below.

Abstractions

- Elements
- Multiplicities

Basic

EMOF merges InfrastructureLibrary::Core::Basic from UML 2 Infrastructure. The results of the merge are given in the following diagrams. The results of merging the capabilities described in the next section are also shown in some of the diagrams.

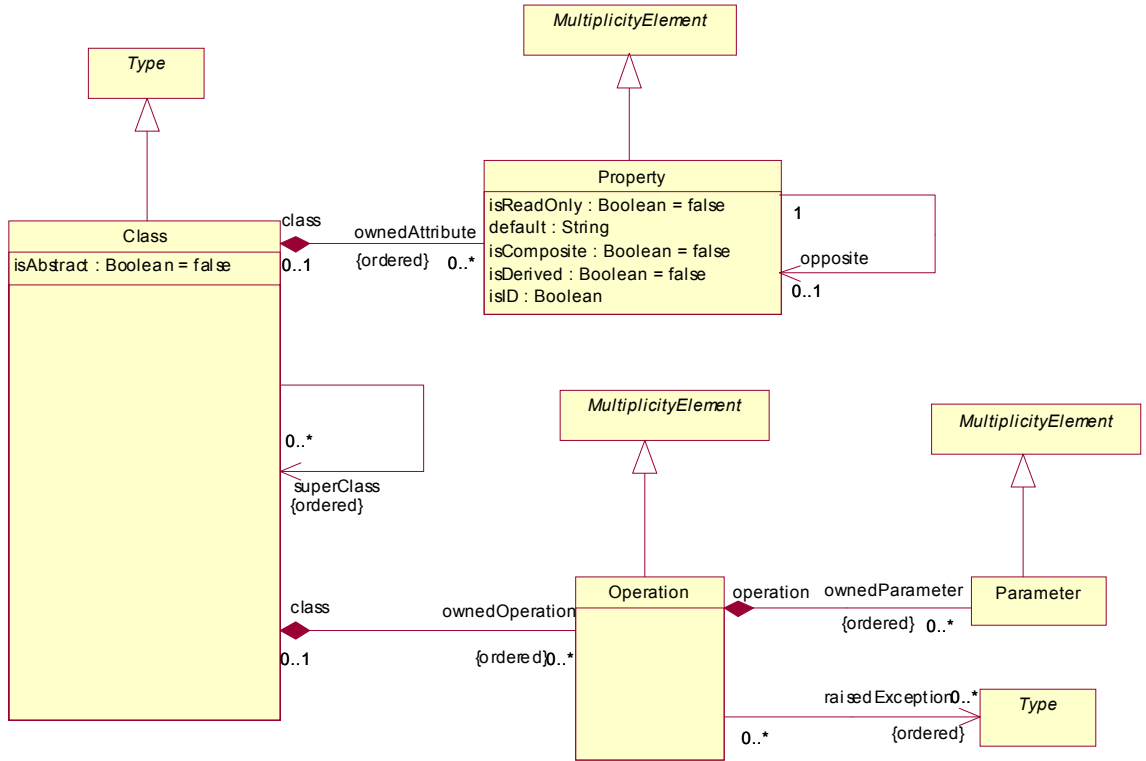


Figure 8 - EMOF Classes

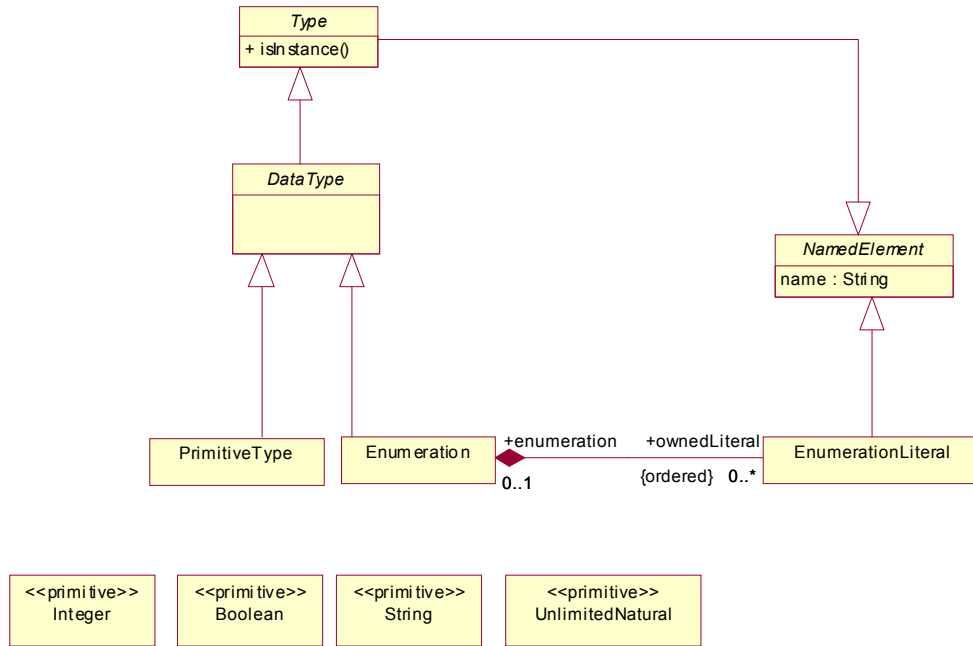


Figure 9 - EMOF Data Types

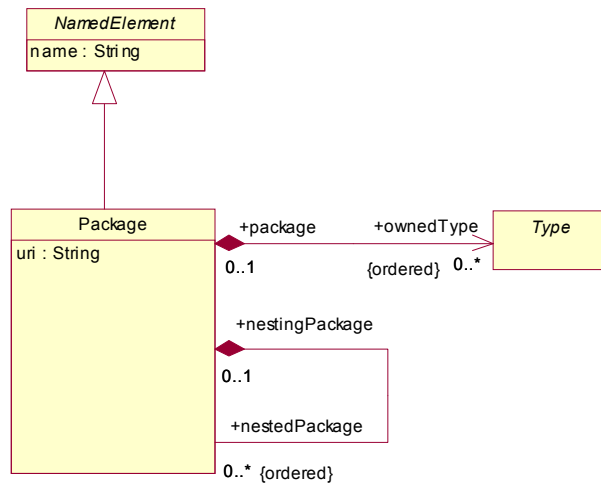


Figure 10 - EMOF Packages

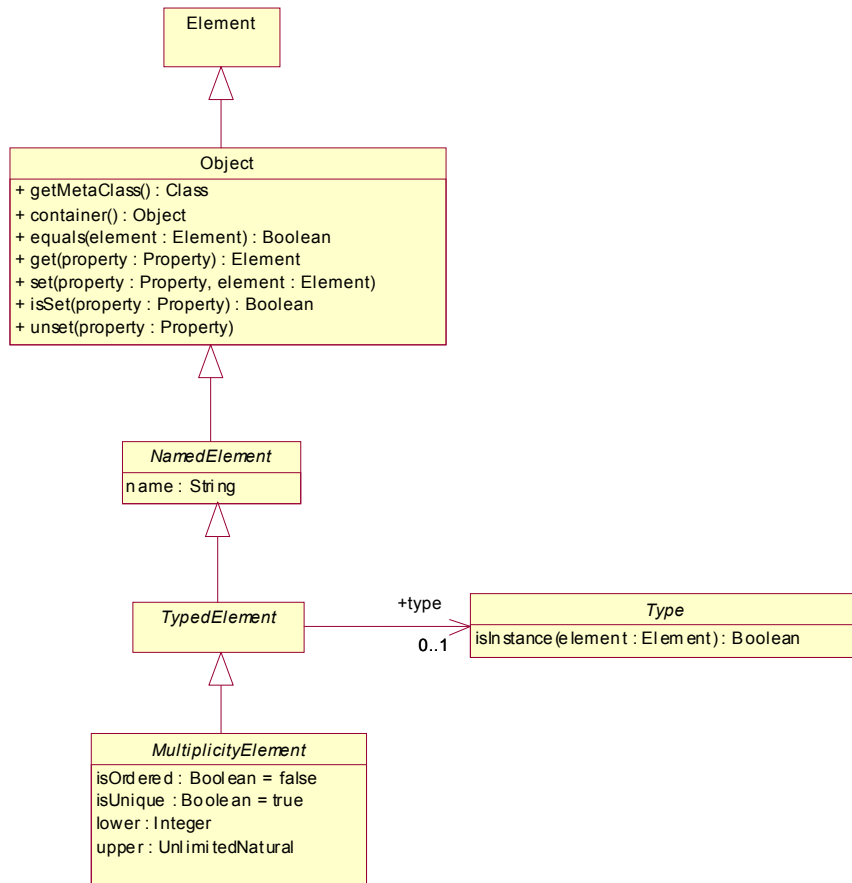


Figure 11 - EMOF Types

12.5 Merged Elements from MOF

The EMOF Model merges the following packages from MOF. See the capabilities chapters 4 through 7 for diagrams of the EMOF capabilities. The results of merging some of these capabilities is shown in the diagrams above.

- Identity
- Reflection
- PrimitiveTypes
- Extensions

12.6 EMOF Constraints

[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 Types and Properties (though there is nothing to prevent these names being automatically generated by a tool).
- [4] Core::Basic and EMOF does not support visibilities. All property visibilities expressed in the UML 1.4 MOF model will be ignored (and everything assumed to be public). Name clashes through names thus exposed should be avoided.
- [5] The definitions of Boolean, Integer, and String are consistent with the following implementation definitions:
 - Boolean: <http://www.w3.org/TR/xmlschema-2/#boolean>
 - Integer: <http://www.w3.org/TR/xmlschema-2/#integer>
 - String: <http://www.w3.org/TR/xmlschema-2/#string>
- [6] All the abstract semantics specified in the Chapter 1, “CMOF Abstract Semantics” are optional for EMOF.
- [7] X is an object and therefore supports reflection if `MOF::Object.isInstance(X)==true`

12.7 EMOF Definitions and Usage Guidelines for the Basic Model

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`.
- “Identity” 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 [SICSteve CookList again](#) 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.

Element

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

13 CMOF Reflection

CMOF Reflection provides extended capabilities over the EMOF Reflection package. The relationships are as follows:

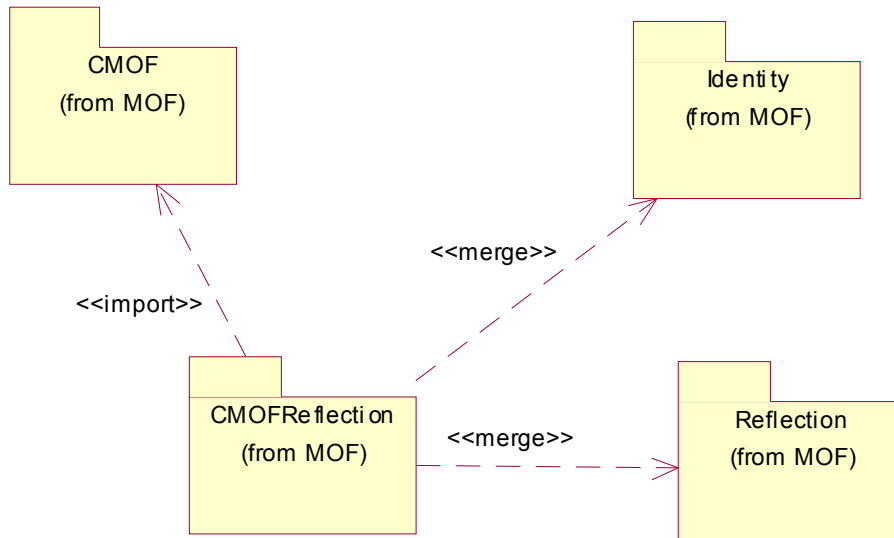


Figure 12 - CMOF Relations

Note that CMOF Reflection merges operations into the existing Object, Extent and Factory classes, and adds a Link class and an Argument datatype. The additions in CMOF Reflection are as follows:

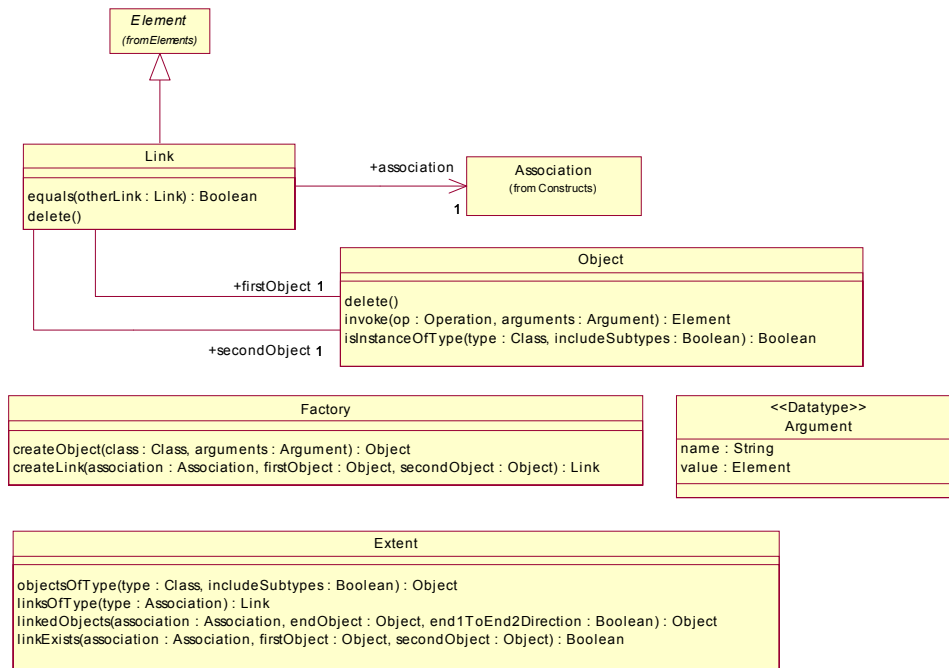


Figure 13 - CMOF Reflection package

13.1 Link

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

Properties

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

firstObject: Object This is the Object associated with the first end of the Association.

secondObject: Object This is the Object associated with the second end of the Association.

Operations

equals(otherLink:Link): Boolean

Returns True iff the otherLink has association, firstObject, secondObject all equal to those on this Link.

delete()

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

Constraints

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

The secondObject 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 the link is created, it is not assigned to any Extent.

Rationale

Since MOF 2.0 allows the same pair of objects 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 Object.

Changes from MOF 1.4

None. The MOF 1.4 navigation capabilities are retained.

13.2 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 Objects and data values to be supplied.

Properties

name: String The name of the argument.

value: Element The value of the argument.

Constraints

None: constrains will be dependent on the context of where the Argument is supplied.

Semantics

None.

Rationale

Since MOF 2.0 allows Operation parameters and Properties to have defaults, it is necessary to explicitly identify the values supplied.

Changes from MOF 1.4

Values supplied to constructors are now identified rather than being deduced through the ordering.

Object

CMOF Reflection adds the following extra operations.

Operations

delete()

Deletes the Object.

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

Calls the supplied Operation on the object, passing the supplied Arguments and returning the result.

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.

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

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

Rationale

Adds the equivalent of MOF 1.4 capabilities.

Changes from MOF 1.4

Parameters to operations are now identified by name.

13.3 Factory

CMOF Reflection adds two extra operations.

Operations**createObject(class:Class, arguments : Argument[0..*]) : Object**

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, firstObject : Object, secondObject : Object) : Link

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

Rationale

Adds the equivalent of MOF 1.4 capabilities.

Changes from MOF 1.4

Names have changed.

13.4 Extent

CMOF Reflection adds four extra operations.

Operations**objectsOfType(type : Class, includeSubtypes : Boolean) : Object[0..*]**

This returns those objects 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) : Link[0..*]

This returns those links in the extent that are instances of the supplied Association.

linkedObjects(association : Association, endObject : Object, end1ToEnd2Direction : Boolean) : Object[0..*]

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

linkExists(association : Association, firstObject : Object, secondObject : Object): Boolean

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

Rationale

Adds the equivalent of MOF 1.4 capabilities.

Changes from MOF 1.4

Names have changed.

14 The Complete MOF (CMOF) Model

The CMOF Model is the metamodel used to specify other metamodels such as UML2. It is built from EMOF and the Core::Constructs of UML 2. The Model package does not define any classes of its own. Rather, it merges packages with its extensions that together define basic metamodeling capabilities.

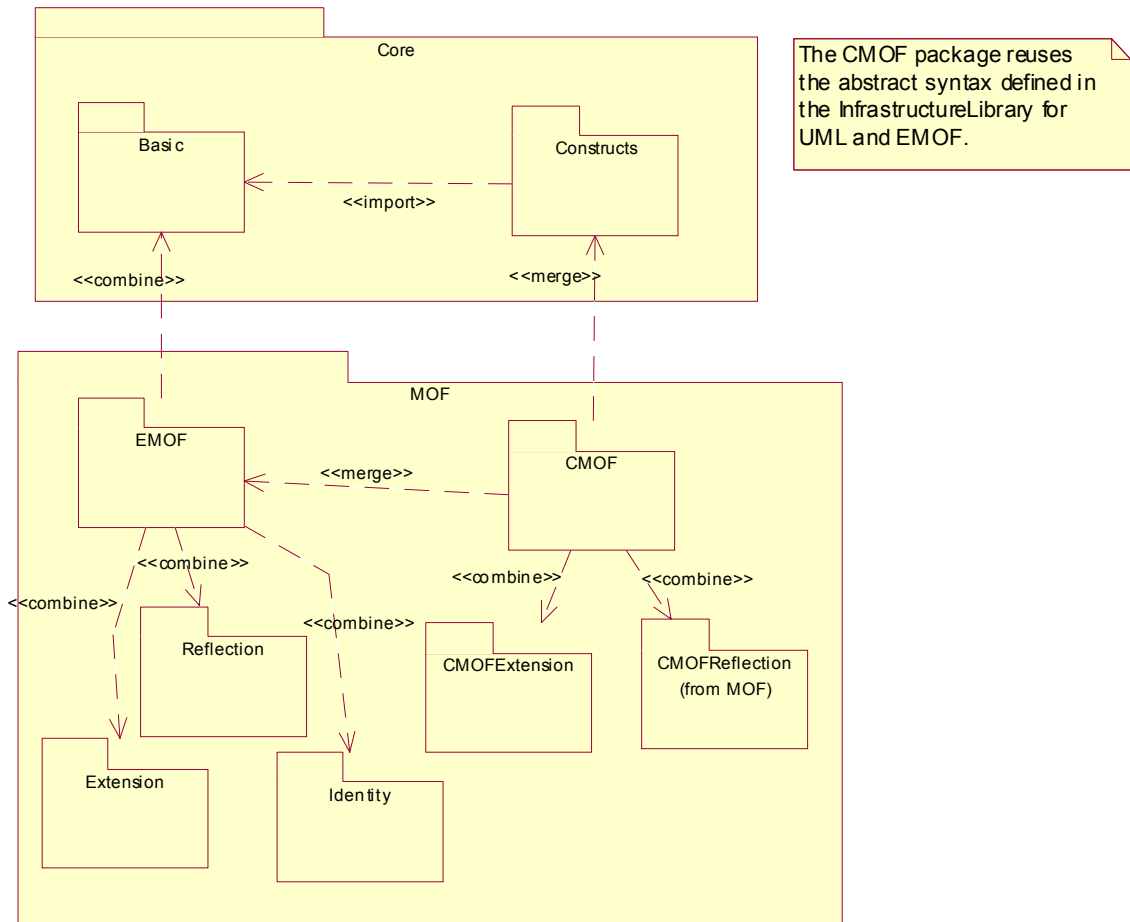


Figure 14 - CMOF Packages

14.1 CMOF Extensions to Core::Constructs

CMOF extends Core::Constructs to specify different package merge semantics.

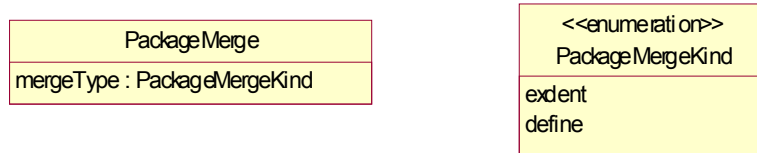


Figure 15 - CMOF Extensions to Core::Constructs

14.2 PackageMerge

CMOF extends Core::Constructs::PackageMerge with the ability to specify different kinds of package merge semantics. A package merge can either define a new package consisting of the merging package and containing all the model elements of the merged package, or it can extend the merged packages with capabilities defined in the merging package. The semantics of PackageMergeKind::extend are given in “Unified Modeling Language: Infrastructure”, version 2.0, section 5.9 Packages diagram, class PackageMerge.

A package merge with PackageMergeKind::define is a relationship between two packages, where the contents of the target package (the one pointed at) is merged with the contents of the source package through deep copy, where applicable. Contrast with PackageMergeKind::extend as defined in Chapter 5 of “Unified Modeling Language: Infrastructure” which uses inheritance and redefinition to merge packages.

This is a mechanism that should be used when elements of the same name are intended to represent the same concept, regardless of the package in which they are defined, but it is desirable to avoid coupling with the merged packages and the inheritance hierarchies and redefinitions that result from PackageMergeKind::extend. A merging package will take elements of the same kind with the same name from one or more packages and merge them together into a new, single element that does not inherit from, or redefine in any way, the matching merged element.

It should be noted that a package merge can be viewed as a short-hand way of explicitly defining new packages. The merged packages are still available, and the elements in those packages can be separately qualified. But there is no type conformance between matching merged and merging elements.

From an XMI point of view, it is either possible to exchange a model with all PackageMerges retained or a model where all PackageMerges have been transformed away (in which case package imports, generalizations, and redefinitions are used instead).

Properties

mergeType: PackageMergeKind Specifies the kind of package merge to perform, define, or extend.

Operations

No additional operations.

Constraints

No additional constraints.

Semantics

A package merge with `PackageMergeKind::define` between two packages implies a set of transformations, where the contents of the merged package is copied into the merging package. Each element has its own specific expansion rules. The package merge dependency is removed from the model since there is no longer any need for references of any kind between the packages.

An element with private visibility in the merged package is not expanded in the merging package. This applies recursively to all owned elements of the merged package.

A Package can directly contain PackageableElements which include subpackages, classes, associations, OCL constraints, etc. PackageableElements may in turn have ownedElements of their own. When packages are merged, all ownedElements are recursively merged as follows. Every ownedElement of the merged target that has no matching ownedElement in the same parent element in the merging source is copied using a deep copy (i.e., recursively copies all its ownedElements) into the merging source. If the merged element matches a merging element, then rules specific to the element metatype are applied to merge the matching elements together into a single new element.

Package PackageMergeKind::define Rules

- Packages match by name. Each nested subpackage creates a new namespace with respect to matching.
- Non-matching ownedElements of the merged package are copied into the merging package.
- The merges of matching ownedElements of the merged and merging packages are merged according to the rules specified for the element metatype.
- If any matching ownedElements cannot be merged, then the merge is not well formed.

Class PackageMergeKind::define Rules

- Classes match by name.
- Non-matching properties of the merged target class are copied into the merging source class.
- The merges of matching properties are included in the merging source class according to the Property merge rules.

Property PackageMergeKind::define Rules

- Properties match by name.
- The merging source property is that property that has the most specific type conforming to the merging and merged property types in the merging package, and with the most specific multiplicity.
- The most specific type is that type in the merging package that minimally conforms (is a supertype of) the types of the merged and merging properties. Note that the most specific type may be a type that is created as the result of another merge, and not directly a type of either the merged or merging properties.
- The most specific multiplicity has the larger lower bound and smaller upper bound of the merged and merging properties.
- If the merged and merging properties have non-conforming types, or there is no most specific multiplicity, then the merge is not well formed. Note that the merging package can introduce new types that can be used to provide a conforming type in necessary, but this is not done automatically by the merge semantics.

Association PackageMergeKind::define Rules

- Associations match by name if they are named. Otherwise they match if their memberEnd Properties match.
- Non-matching Associations of the merged target package are copied into the merging source package.
- Matching Associations are merged by class and property merge rules.

Operation PackageMergeKind::define Rules

- Operations match by name and Parameter type except returnResult parameters. ReturnResult parameters are not included because they can result in ambiguous operation invocations.
- Non-matching Operations of the merged target Class are copied into the merging source Class.
- Matching Operations are ignored.

Rationale

The additional PackageMergeKind::define merge semantics are required to produce EMOF as a merge of Core::Basic without introducing lots of multiple inheritance and redefinitions. It may also be generally useful for creating new metamodels from parts of existing metamodels where inheritance and coupling to those merged metamodels is not desirable.

Changes from MOF 1.4

MOF 1.4 did not support any form of package merging.

14.3 Imported Elements from UML 2 Core

By merging the Constructs package from the UML 2 Core, the CMOF Model's members include the following packages from Abstractions as well as Basic, as listed below.

Abstractions

- Namespaces
- Classifiers
- Comments
- Constraints
- Ownerships
- Multiplicities
- Relationships
- StructuralFeatures
- BehavioralFeatures
- Visibilities
- Super
- Changeabilities

- Redefinitions
- Expressions
- TypedElements

Figure 16 shows some of the key concrete classes and associations of the UML 2 Core imported by MOF Model. There are many other important elements in the Core, but these provide the structure of class modeling.

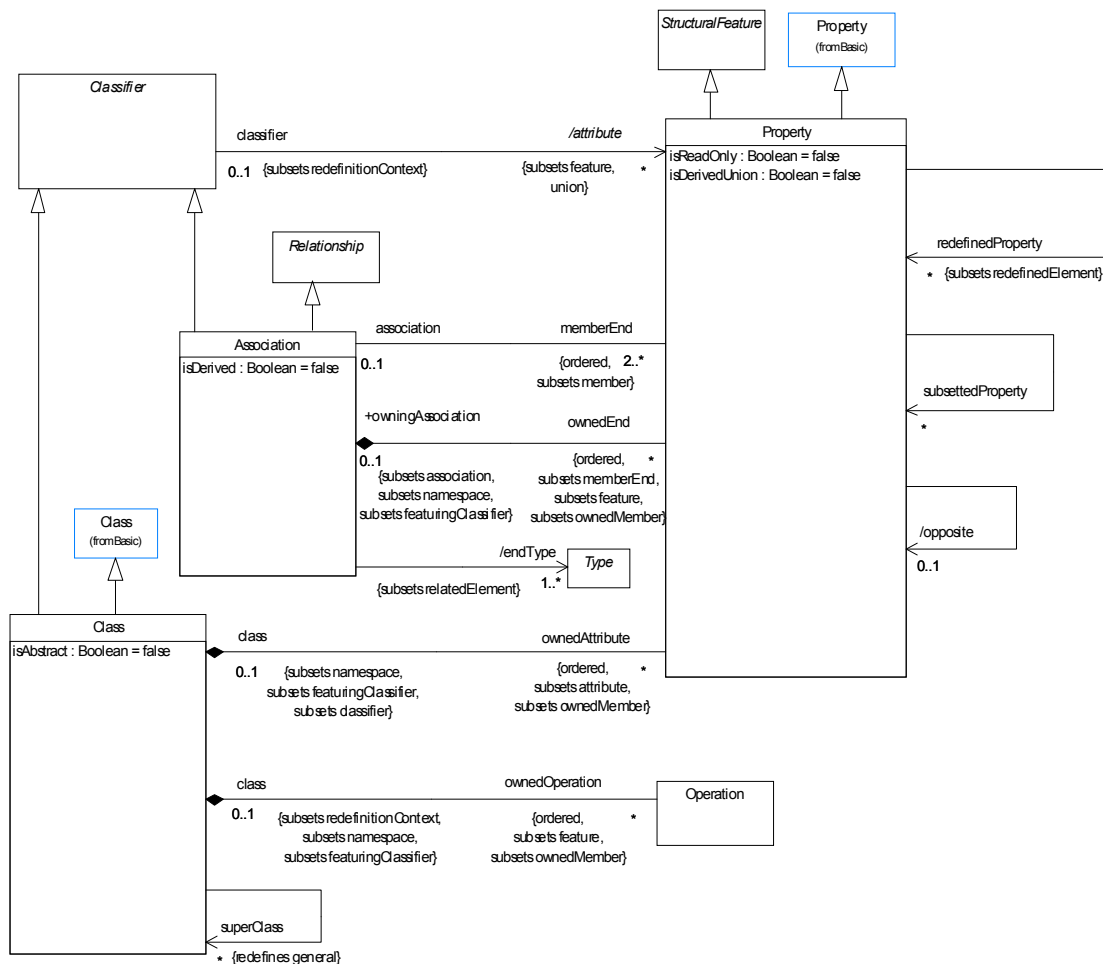


Figure 16 - Key concrete classes from the UML Core::Constructs

14.4 Imported Elements from MOF

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

- Identity

- Reflection
- Extension

14.5 CMOF Constraints

This section details additional constraints owned by the CMOF package which are applied to metamodels to be processed by a CMOF implementation.

- [1] The multiplicity of `Association::memberEnd` is limited to 2 rather than $2..*$ (i.e. n-ary Associations are not supported).

```
context Association inv: memberEnd->size() = 2
```

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

```
context Operation inv: raisedException.oclIsType(Class)
```

- [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 $2^{31} - 1$.

```
context Integer inv: value >= -231 and value <= 231 - 1
```

- [4] In order to support limited implementations of the String class, each instance of String occurring as an attribute value of an element is 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 classifiers and features (though there is nothing to prevent these names being automatically generated by a tool).

```
context Classifier inv: not(name = OclUndefined)
context StructuralFeature inv: not(name = OclUndefined)
```

- [7] Visibilities will be ignored (and everything assumed to be public). Name clashes through names thus exposed should be avoided.

- [8] Enumerations may not have attributes or operations

```
context Enumeration inv: isEmpty(feature)
```

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

14.6 CMOF Extensions to Capabilities

This section details CMOF extensions to the MOF2 capabilities.

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 for default values for structured data types.

Extension

- [1] CMOF extends package Extension with role name “tag” for the non-navigable end on the association between Tag and Element.

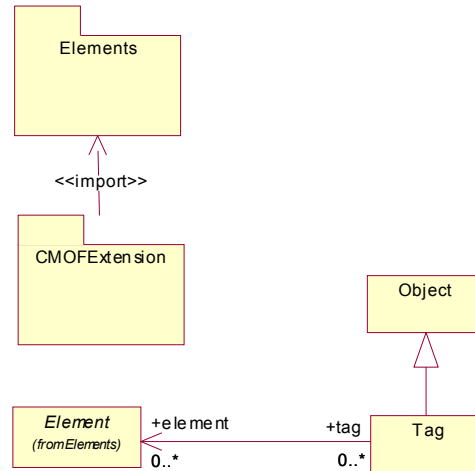


Figure 17 - CMOF Extension Packages

Part IV - Abstract Semantics

This part of the MOF 2 specification addresses abstract mappings from the MOF model: This forms the foundation for defining concrete mappings to various technologies such as XMI and JMI.

15 CMOF Abstract Semantics

This chapter 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, settability, 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).

15.1 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 section 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 2.0 series. It is not the intention of this section to mandate that all MOF implementations need support all of these capabilities: more to define what the capabilities mean when they are provided. Compliance points are described separately. For example, this section 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 2.0 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 section 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 UML2 Infrastructure on which it’s 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.

15.2 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 behaviour (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
- retain the ‘illusion’ that these properties are true attributes
- 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).

The following represents the Semantic Domain model for Constructs. It extends the abstract syntax for Instances.

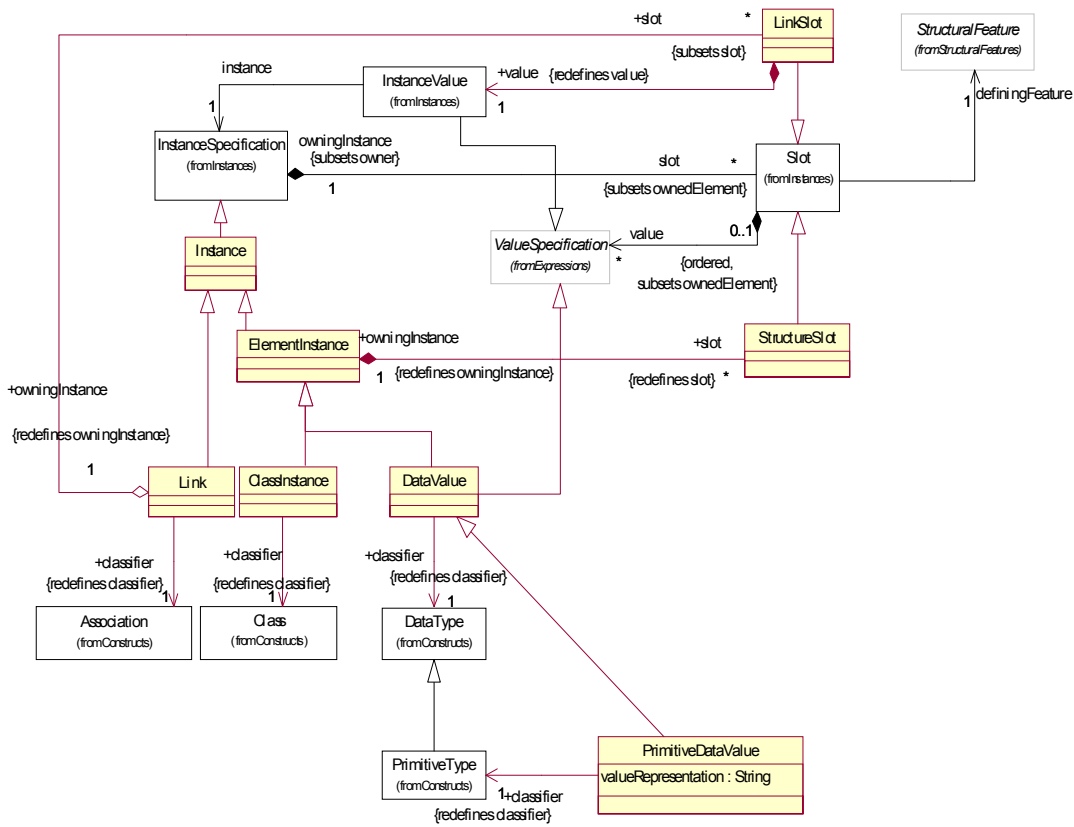


Figure 18 - Semantic Domain model for Constructs

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 stored StructuralFeatures include inherited ones but exclude:
 - -Derived properties (including derived unions)
 - -Properties which have been redefined (there exists a property which is a member of the classifier which has redefinedProperty= this)
2. The classifier is not abstract
3. The instance obeys Constraints which have its classifier as context

ClassInstance

1. An instance is owned via at most one composition
2. At most one Slot for an isComposite property may have a value. (this needs more work if the owner reference is not navigable)
3. Compositions are not cyclic

Slot

1. Each value is compatible with the type of the definingFeature

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 (it is owned by a Class) 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 `subsettingProperty` may be set

Association

An Association is derived iff all its Properties are derived.

15.3 Notes

This section 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: inbuilt 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 is 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: these are defined in 10.8.

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

15.4 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::verify() modeled as **ObjectInstance::verify()**

-- Check all cardinalities and constraints (for CMOF classes only)

post: (self.allProperties->forall(p | let v = self.get(p) and

(p.lower <> null implies v.size() >= p.lower) and

(p.upper <> null implies v.size() <= p.upper) and

(p.isUnique implies v.size() = asSet(v).size) and

(self.classifier.container = EMOF implies

(p.isReadOnly implies not(self.isSet(p)))

and

(self.classifier.container = CMOF implies

self.classifier.allConstraints->forall(c| self.evaluateConstraint(c))

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})

-- Issue: should defaults be substituted for the parameters before extInvoke? - probably yes

post: result = extInvoke(op, args)

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)

15.5 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)
```

15.6 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) which 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(association.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::convertToString(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!

15.7 Extent Capabilities

This section describes minimal capabilities for MOF Core. It does not address issues such as extent creation that are in scope of MOF 2.0 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:

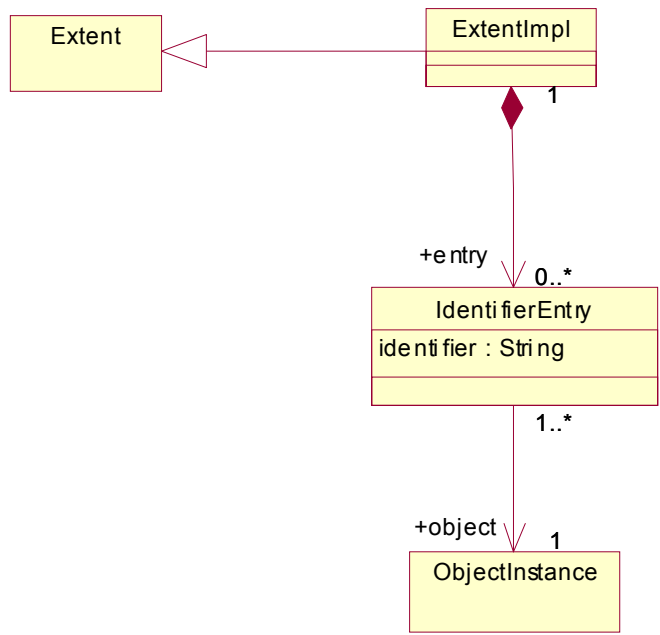


Figure 19 -

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
```

15.8 Additional Operations

- [1] This gives all of the properties in the namespace of the class (including inherited) that require a slot: it excludes properties owned by Associations, derived properties and those that subset or redefine another property (in which case that property will provide the slot and the redefinition will just restrict the values)

Class::allSlottableProperties(): Set(Property);

```
allSlottableProperties = member->select(p| p.oclsKindOf(Property) and
    not (self.allParents() includes p.namespace)
    -- excludes foreign association ends
```

not(p.isDerived) and
isEmpty(p.redefinedProperty) and
isEmpty(p.subsettedProperty))

- [2] This returns the slot corresponding to the supplied property. For redefining properties it will be the redefined one. Note that derived properties will only have slots if the redefine a non-derived one so the result may be null.

Instance::propertySlot(Property p): Slot
propertySlot = self.slot->select(definingFeature = p.originalDefinition))

- [3] This returns the original definition of a Property through any number of redefinitions and subsetting.

Property::originalDefinition(): Property
post: (isEmpty(redefinedProperty) and isEmpty(subsettedProperty) and result = self) or
(notEmpty(redefinedProperty) and result = redefinedProperty.originalDefinition()) or
(notEmpty(subsettedProperty) and result = subsettedProperty.originalDefinition())

- [4] This returns the single Property that represents the current owner of the Object based on current instance values; may be null for top level objects

Object::owningProperty(): Property
post: result = self.allProperties->select(op| op.isComposite and self.get(op) <> null)

- [5] 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)

- [6] This returns all the Constraints for a classifier

CMOF::Classifier::allConstraints(): Set(Constraint)
post: result = extent().objectsOfType(Constraint)->select(c | c.context = self)

- [7] 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))

[8] 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))

[9] 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)

[10] 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)

[11] 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)

[12] 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))

[13] 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}})

16 Migration From MOF 1.4

This chapter addresses the migration of existing MOF 1.4 metamodels to MOF 2.0 Complete. MOF 1.4 metamodels can be translated to MOF 2.0 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 2.0 NamedElement::name) are not listed here.

16.1 Metamodel Migration

A valid MOF 1.4 metamodel can be translated to a MOF 2.0 model. Translation is based on a straightforward mapping that can be fully automated.

MOF 1.4	MOF 2.0 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” (<i>pending proposed change to U2P</i>)
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 2.0 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. If the property is owned by the association, it is not navigable (no reference is referencing it). 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 2.0 all the ends are navigable in terms of MOF 1.4 navigability. MOF 2.0 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 revisited in conjunction with the MOF 2.0 Facility RFP Note: For importing individual elements inside packages Import maps to ElementImport. Note that in MOF 2.0, an import makes names within imported packages visible without qualification.
Import::importedNamespace	PackageImport::importedPackage, PackageMerge::mergedPackage, ElementImport::importedElement Note that in MOF 2.0, an import makes names within imported packages visible without qualification
Tag	Extension::Tag

16.2 API Migration

This section summarizes the API equivalents between the MOF 1.4 Reflective API, the JMI API and the proposed MOF 2.0 Core API. Class names are in bold and italics.

Note that many of the gaps for MOF 2.0 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 2.0
<i>RefBaseObject</i>	<i>RefBaseObject</i>	<i>Object</i>
refMofId	refMofId	Extent::identifier
refMetaObject	refMetaObject	getMetaclass
refImmediatePackage	refImmediatePackage	
refOutermostPackage	refOutermostPackage	
refItself	<i>Java Object.equals</i>	Object::equals
refVerifyConstraints	refVerifyConstraints	verify
refDelete	RefObject::refDelete	delete
<i>RefObject</i>	<i>RefFeatured</i>	
refValue	refGetValue	get
refSetValue	refSetValue	set
refUnsetValue	<i>Set value to null</i>	unset
refAddValueBefore	<i>Use of live collections</i>	
refAddValueAt	<i>ditto</i>	
refModifyValue	<i>ditto</i>	
refModifyValueAt	<i>ditto</i>	
refRemoveValue	<i>ditto</i>	
refRemoveValueAt	<i>ditto</i>	
refInvokeOperation	refInvokeOperation	invoke
<i>Test for value = null</i>	<i>Test for value = null</i>	isSet
	<i>RefObject</i>	
refIsInstanceOf	refIsInstanceOf	isInstanceOfType
	refClass (gets proxy)	
refImmediateComposite	refImmediateComposite	container
refOutermostComposite	refOutermostComposite	
RefBaseObject::refDelete	refDelete	delete
<i>RefClass</i>	<i>RefClass</i>	
refCreateInstance	refCreateInstance	Factory::create

refAllObjects (includeSubtypes=true)	refAllOfType	Extent::objectsOfType (includeSubtypes=true)
refAllObjects (includeSubtypes=false)	refAllOfClass	Extent::objectsOfType (includeSubtypes=false)
	refCreateStruct	Factory::createFromStruct
	refGetEnum	Factory::createFromEnum
refAssociation	refAssociation	
refAllLinks	refAllLinks	Extent::linksOfType
refLinkExists	refLinkExists	Extent::linkExists
refQuery	refQuery	Extent::linkedObjects
refAddLink	refAddLink	Factory::create
refAddLinkBefore	<i>Use live collections</i>	
refModifyLink	<i>ditto</i>	
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)	<i>Java Object.equals</i>	
refClassRef	refClass	
refAssociation	refAssociation	
refPackage	refPackage	
	refAllPackages	
	refAllAssociations	
	refCreateStruct	
	refGetEnum	
RefBaseObject::refDelete	refDelete	
		useContainment (attribute)
		objects

		identifier
		object

Part V - Appendices

The appendices for this document include:

- Appendix A - XSD and XMI for MOF 2.0
- Appendix B - JMI Java Language Mapping

A XSD and XMI for MOF 2.0

The Meta Object Facility (MOF) 2.0 XMI Mapping Specification : ptc/03-10-09 defines extensions to XMI 2.0. Document ad/2003-04-08 contains the XSD and XMI files for MOF 2.0.

B JMI Java Language Mapping

The Java language mapping for MOF 1.4 defined by the JSR-40 Java Metadata Interface (JMI) Specification expert group as part of the Java Community Process will require revision to support MOF 2. The Java language mapping for MOF 2.0 will be defined using a new JSR under the Java Community Process.

