**Date**: April 2013


# MOF Support for Semantic Structures

*Version 1.0*

---

OMG Document Number: formal/2013-04-02
Standard document URL: http://www.omg.org/spec/SMOF/1.0/
Machine Consumable Files:
Normative: http://www.omg.org/spec/SMOF/20120801/SMOF.xmi
http://www.omg.org/spec/SMOF/20120801/SMOFAbstractSemantics.ocl
Non-normative: http://www.omg.org/spec/SMOF/20120801/MOF-SMOF.zip
http://www.omg.org/spec/SMOF/20120801/SMOFAbstractDomainModel.emx

---

## 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, 109 Highland Avenue, Needham, MA 02494, U.S.A.

## TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , IMM™, MOF™ , OMG Interface Definition Language (IDL)™ , and OMG Systems Modeling Language (OMG SysML)™ 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.

# OMG's Issue Reporting Procedure

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

# Table of Contents

# Preface

## About the Object Management Group

### OMG

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

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

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

## OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

*http://www.omg.org/spec*

Specifications are organized by the following categories:

### Business Modeling Specifications

### Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

### IDL/Language Mapping Specifications

### Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

### Modernization Specifications

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

- **CORBAServices**
- **CORBAFacilities**

### OMG Domain Specifications

### CORBA Embedded Intelligence Specifications

### CORBA Security Specifications

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


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


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

## Typographical Conventions

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

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

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

**`Courier - 10 pt. Bold:`** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

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

## Issues

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

# 1 Scope

The Meta Object Facility has proven itself as a valuable and powerful foundation for a family of modeling languages, like UML, ODM, CWM, etc.

However, MOF 2 suffers from the same structural rigidity as many object-oriented programming systems, lacking the ability to classify objects by multiple metaclasses, the inability to dynamically reclassify objects without interrupting the object lifecycle or altering the object's identity, and a too constrained view on generalization and properties.

This extension to MOF modifies MOF 2 to support dynamically mutable multiple classifications of elements and to declare the circumstances under which such multiple classifications are allowed, required, and prohibited.

# 2 Conformance

The Semantic MOF specifies two compliance options:

1. SMOF for CMOF
2. SMOF for EMOF

## 2.1 SMOF for CMOF Compliance

As described in Clause 9, package merge is used to extend the CMOF metamodel to produce the SMOF for CMOF, or SCMOF compliance level.

## 2.2 SMOF for EMOF Compliance

As described in Clause 9, package merge is used to extend the EMOF metamodel to produce the SMOF for EMOF, or SEMOF compliance level. This also necessitates the inclusion of Abstractions::Constraints and Abstractions::Expressions into SEMOF, because Semantic MOF of its nature involves the declaration of constraints.

# 3 Normative References

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

## 3.1 List of Normative References

Meta Object Facility (MOF) Core Specification, Version 2.4.1, OMG Document formal/2011-08-07

Meta Object Facility (MOF) Facility Object Lifecycle, Version 2.0, OMG Document formal/10-03-04

OMG Unified Modeling Language™ (UML), Superstructure, Version 2.4.1, OMG Document formal/2011-08-06

MOF/XMI Mapping, Version 2.4.1, OMG Document formal/2011-08-09

The Object Constraint Language (OCL) Version 2.3.1 is used to define constraints and semantics in subsequent clauses of this specification. The OCL 2.3.1 language definition can be found here:

Object Constraint Language Specification, Version 2.3.1, OMG Document formal/2012-01-01

## 3.2 List of Non-Normative References

The following specification is mentioned in descriptive text of subsequent clauses, but does not constitute a normative part of this specification:

- Semantics of a Foundational Subset for Executable UML Models, Version 1.0, OMG Document formal/2011-02-01

# 4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

| | |
|---|---|
| Multiple Classification | The type of an object resulting from instantiating the union of structural and behavioral features defined by two or more independent metaclasses into a single object. |
| Dynamic Reclassification | The ability to add or remove metaclasses from the type of an object during the lifecycle of that object. The addition or removal of metaclasses may alter the structure and/or behavior of the object, but does not alter the object's identity. |

# 5 Symbols

No symbols are defined by this specification.

# 6 Additional Information

## 6.1 How to Read this Specification

This specification is part of the MOF 2 specifications. As such, it does not contain a complete specification of the Meta Object Facility version 2, but an increment to extend the MOF 2 Core with features to handle semantic structures. To obtain a complete extended MOF 2 specification, the content of this specification must be merged with the MOF 2 Core specification.

Clause 7 provides several non-normative use cases and examples to introduce the problem area addressed by this specification. Clause 8 formally positions this specification in relationship to the Complete MOF (CMOF) specification contained in the MOF 2 Core document. Clause 9 provides the abstract syntax and detailed descriptions of the MOF extensions specified in this document. Clause 10 provides the corresponding changes to the abstract semantics. Clause 11 contains the required changes to the XMI serialization.

## 6.2 Changes to Adopted OMG Specifications

This specification amends / modifies the following OMG specifications:

- MOF Core 2.4.1
- MOF Facility Object Lifecycle 2.0

## 6.3 Acknowledgements

The following companies submitted this specification:

- 88solutions
- Adaptive
- Deere & Company
- Mega International
- Microsoft
- Model Driven Solutions
- Thematix Partners (formerly known as Sandpiper Software)

The following companies supported this specification:

- Computer Science Corporation

# 7 Concept Overview and Use Cases (informative)

## 7.1 Overview

The Meta Object Facility (MOF) takes a central architectural role in the family of modeling languages developed at the Object Management Group (OMG). The combination of multiple meta-levels and reflection provides a flexible and powerful but simple foundation for more elaborate modeling languages, like UML 2.

However, most object-oriented systems (including MOF) suffer from structural rigidness and lack the ability to address temporal aspects in an elegant way. This makes a correct representation of real-world facts difficult, if not impossible. Problem areas are the type / classification system and object relationships. Currently, if an object is created, it is instantiated with the type and features of its defining class, and it has to live as such until its destruction. In reality, objects are subject to constant variations without changing their identity, they undergo changes in classifications and assumed roles. This deficiency has a direct negative impact on several MOF-based metamodels and languages. Clause 7.2 demonstrates the impact on the *Semantic for Business Vocabularies and Business Rules (SBVR)* specification, and clause 7.3 shows the workarounds needed to base the Ontology Definition Metamodel (ODM) on MOF.

## 7.2 Multiple and Dynamic Classification

In most traditional object-oriented systems, object instances are statically typed (classified) by their defining classifier at instantiation time and retain this type and all structural and behavioral features defined by this type for the entire lifetime of the instance. MOF and most object-oriented programming languages follow this scheme. However, it is often desirable to change the type of an object instance at a certain point of time without affecting the existence (and identity) of the object instance. This is achieved by dynamic classification. The type of an object, ignoring primitive types for a moment, has structural consequences for the object; it is the “blueprint” defining the structural and semantic implementation of the type’s features within this object. If we change the type of an object instance through dynamic classification, then we need to readjust the object’s features according to the new type.

For example, a person identified as “Fred” graduates from university and becomes an employee of a corporation. Consequently we dynamically reclassify Fred from type “Student” to type “Employee.” In this process Fred loses his feature “studentId” and gains the features “employeeNumber” and “monthlySalary.”

Dynamic classification of single-classified objects is rather drastic, as all features of the object need to be revisited. Multiple classification avoids this, the resulting object instances represent the union of feature instances (slots) from all defining classifiers (types) of the object. As a consequence, multiple classified objects become conceptually inhomogeneous; the conceptual single object is structured into slices, where each slice represents the slots defined by one classifier. Applying dynamic classification to multiple classified objects means effectively adding, removing, or exchanging of slices of that object instance without affecting the life of the object as long as at least one slice remains at all time.

Multiple classification of our “Fred” enables him to be an employee during daytime and a student in the evening.

This example provides insight into another interesting side effect of the object slicing caused by multiple classification: Multiple containment. Containment means a “part-of” relationship, which applies on a slice-by-slice basis in multiply classified objects. The classification of our “Fred” as employee implies a part-of relationship to his employing company. But this part-of relationship affects only the employment-relevant features of Fred, which means the slice defined by the “Employee” classification. The same holds for the part-of relationship with the school defined by the “Student” classification. This shows that multiply classified objects may participate concurrently in multiple containments, but on a

slice-by-slice basis to be exact. If a container is deleted, then the corresponding slice is removed from the object. If the object was multiply classified, the corresponding classification is also removed from the object. If the object is singly classified, or if this was the last remaining classification, the object is deleted.

SMOF extends MOF with the abilities of multiple classification and dynamic classification. This is achieved by extending Element in MOF::Reflection with the ability to be classified by more than one metaclass, and by adding new reflective operations to Element which allow querying, adding, and removing of metaclasses during the lifetime of Element. Slices are not explicitly modeled; they are implicitly defined by the features contributed by each classifying metaclass. Slices become visible in the containment and delete semantics, and in the XMI serialization.

## 7.3 Use Case: UML

An example issue with UML is the inability for actor to have the capabilities of a structured classifier.

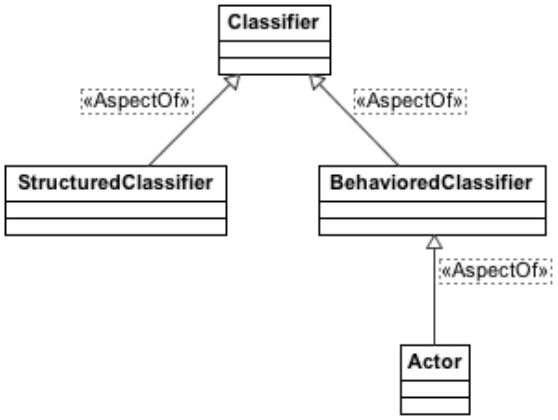


**Figure 7.1**

Consider that Actor, BehavioredClassifer, and StructuredClassifier were aspects as shown above. This would then allow the SAME classifier to be an actor and a structured classifier, yet these concepts remain uncoupled in the metamodel. To allow this capability in the current UML metamodel these all get inherited into a class that could do anything and everything, which makes it unwieldy and difficult to use. It also makes it difficult to add or federate capabilities without modifying the source metamodels. This demonstrates how SMOF facilitates a less coupled approach to metamodeling while allowing a more flexible way to combine features.

## 7.4 Use Case: Semantic of Business Vocabularies and Business Rules (SBVR)

New metamodeling infrastructure layers are being built within ‘MOF’ metamodels: for example the Essential SBVR in the Semantics of Business Vocabulary and Rules (SBVR). The following is an instance diagram example from the *SBVR* specification that shows, to achieve the required flexibility, elements can only be typed by a generic MOF metaclass called Thing. An aim of this specification is to allow SBVR to represent the types of the domain directly in MOF.

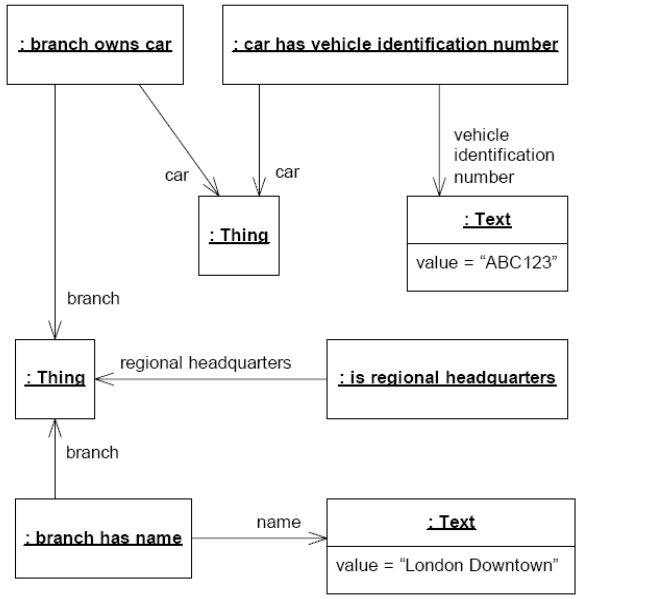


**Figure L.2 - Instance diagram of facts expressed using EU-Rent English Vocabulary**

**Figure 7.2**

## 7.5 Use Case: Ontology Definition Metamodel (ODM)

One of the incentives for this specification was the requirement in OMG specifications for multiple classification. This issue was identified in SBVR as well as "ODM" (Ontology Definition Metamodel). ODM provides a MOF meta model of multiple ontology languages, including OWL. The following model fragment is from ODM.

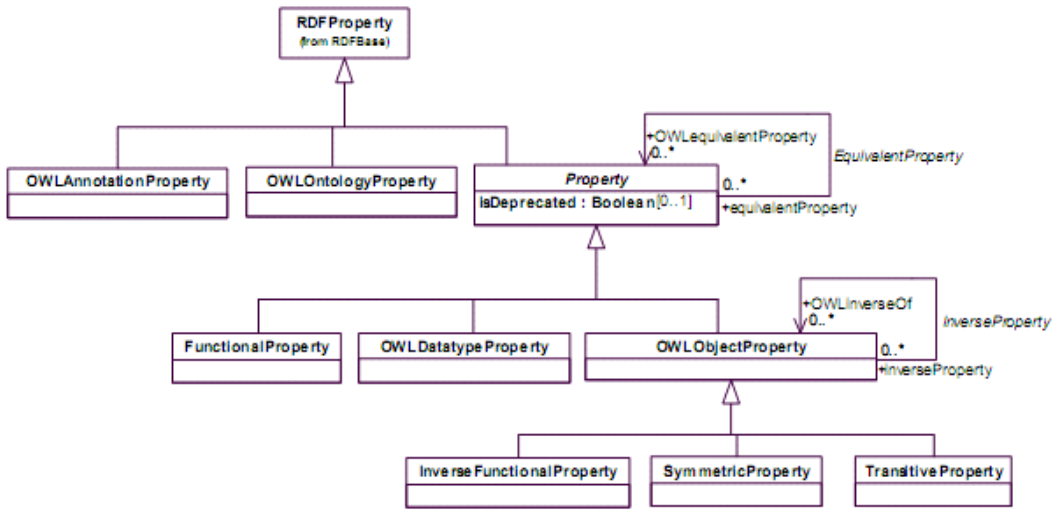


**Figure 7.3**

Note that there are several subclasses of “Property.” This matches the semantics of OWL in that a property can be any of these subclasses but can also be a combination of these classes. A property can, for example, be functional and transitive. Here, due to the single classification restriction of MOF, it is not possible to directly represent the intended OWL semantics or even the OWL structure. In OWL an instance can be classified by any number of classifiers. To allow for the intended OWL semantics in ODM using SMOF, each of the subtypes of Property should be an “AspectOf” of Property and they would then be able to be combined in any order. Where there are restrictions on these combinations “IncompatibleWith” can be used to declare which combinations are invalid.

### Semantic MOF representation of OWL properties

The following model fragment shows the SMOF solution where the generalizations are marked as “aspects” of the more general class. Since each asset is a classification of the same individual this matches the intent of the ODM model without refactoring. Note that some combinations are invalid, which could be represented using “IncompatibleWith” as it is using OWL disjoint.

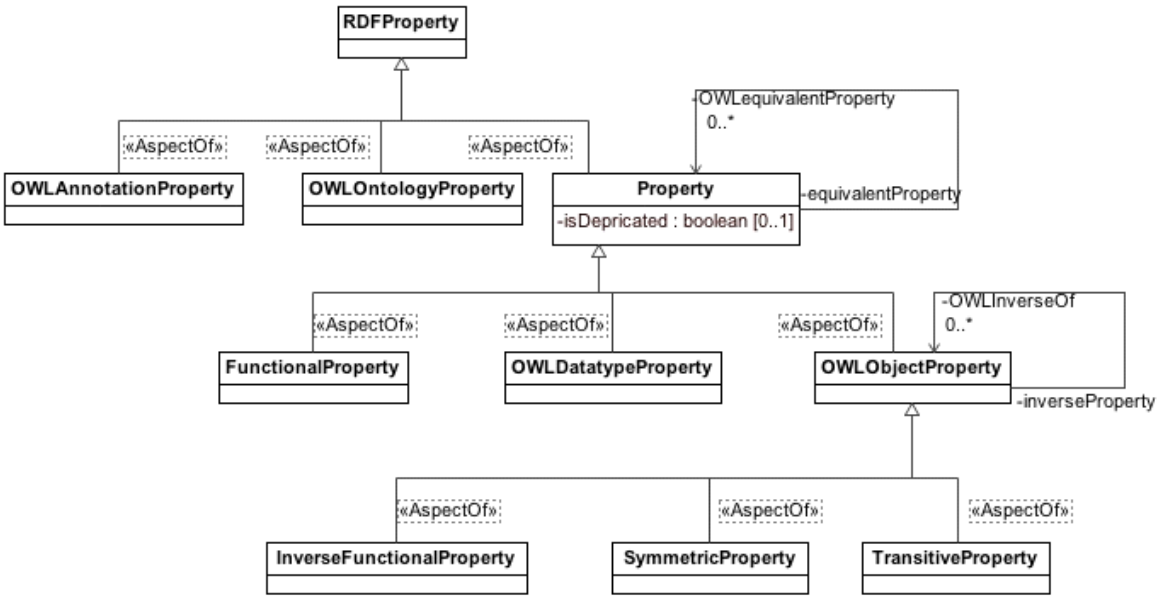


**Figure 7.4**

# 8 Abstract Syntax Architecture

## 8.1 Overview

Semantic structures may be introduced into MOF in multiple ways. However, not every method provides backward compatibility with the existing MOF 2 Core. The approach selected in this specification aims for a maximum of compatibility with MOF 2.

The following diagram shows the SMOF extension of MOF as a Package diagram.

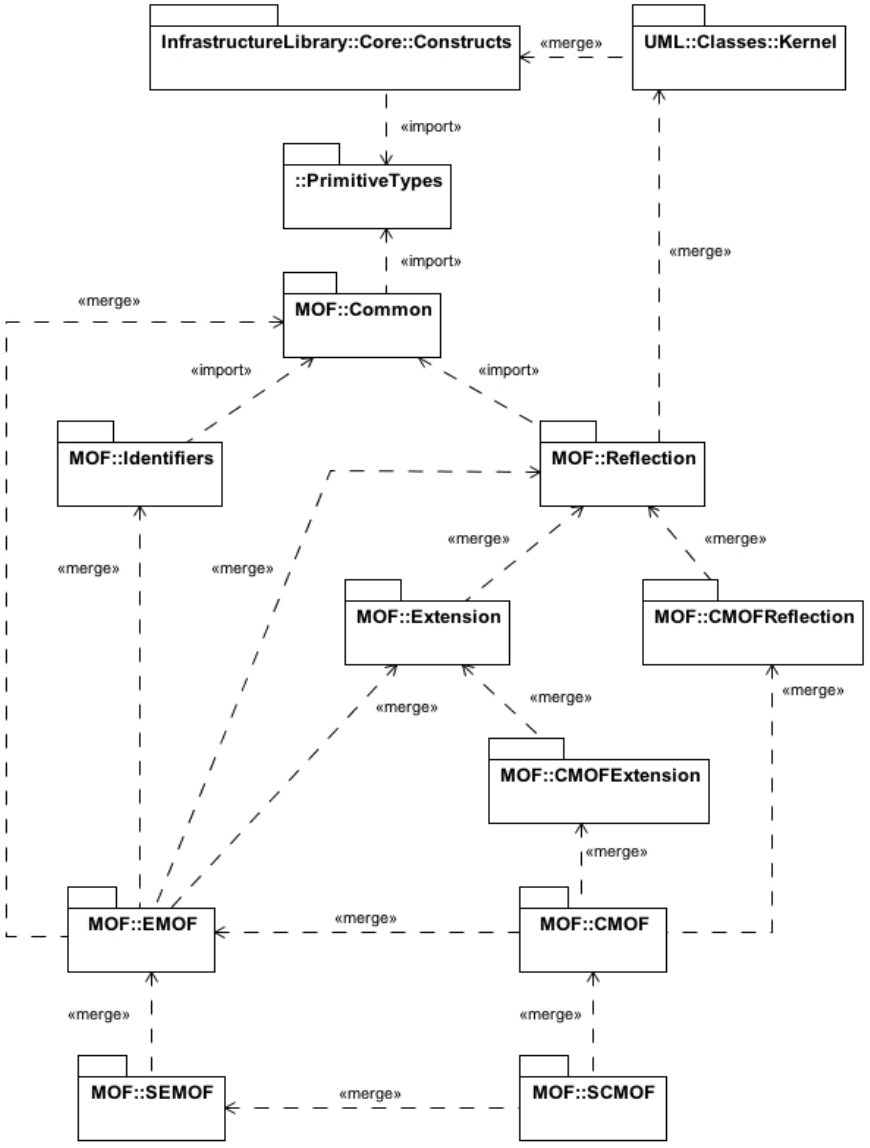


**Figure 8.1 - The SMOF Packages in relation to the EMOF / CMOF Packages**

The SMOF specification is part of the MOF 2 family of specifications. As such, it constitutes an increment building on top of the MOF 2 Core. To obtain a complete extended *MOF 2* specification with support for semantic structures (SMOF), the content of this specification must be merged with the *MOF 2 Core* specification using Package Merge.

In order to support the two SMOF compliance levels, SEMOF as extension of EMOF, and SCMOF as extension of CMOF, additional package merge steps are required due to the limitations of EMOF.

Package SEMOF contains all MOF 2 Core extensions provided by SMOF. Beginning with MOF Core 2.4, MOF shares the metamodel with UML Superstructure by reusing UML's Kernel package. Constraints in the *MOF Core 2.4.1* specification enumerate the concrete metaclasses from UML's Kernel permitted for use by MOF metamodels separately for EMOF and CMOF.

SMOF requires the concrete metaclasses Constraint, Expression, and OpaqueExpression, which are not available in EMOF. Therefore this specification amends constraint [8] in clause 12.4 of the *MOF Core 2.4.1* specification by:

For SEMOF, the following concrete metaclasses from UML's Kernel may also be used:

- Constraint
- OpaqueExpression

Package SCMOF does not contain any SMOF-specific extensions; it merges the additional features of CMOF (compared to EMOF) into package SEMOF.

This specification amends clause 12.5 of the *MOF Core 2.4.1* specification, in the part headed **Property::isComposite==true**, as follows:

Replace "An object may have only one container" by "An object may have at most one container, or if the object is multiply classified, at most one container per classifier."

Replace "Only one container property may be non-null" by "Only one container property, or if the object is multiply classified only one container property per classifier, may be non-null."

Replace "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" by "If an object has an existing container and a new container is to be set for the same container property, the object is removed from the old container before the new container is set."

# 9 Metamodel Extensions

## 9.1 Common SMOF Extensions

### 9.1.1 Abstract Syntax

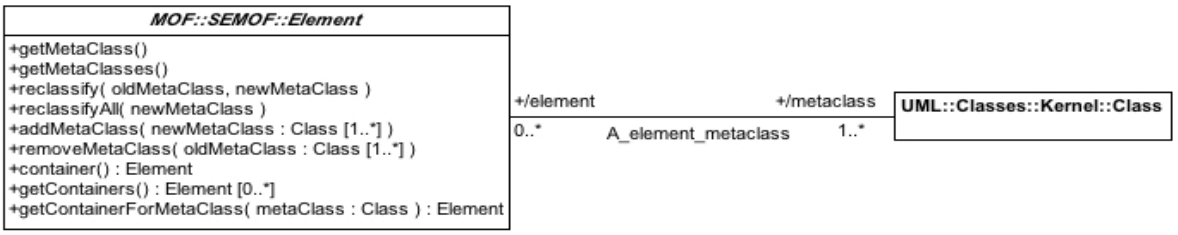


**Figure 9.1 - Reflection, as extended by SMOF**

### 9.1.2 Class Descriptions

#### 9.1.2.1 Constraint (as extended)

**Description**

The semantics of Constraint from MOF are extended to express disjoint and equivalent Classes.

**Semantics**

A Constraint with constrainedElements that are Classes, and with specification that is an OpaqueExpression with language[0] = ‘SMOF’ and body[0] one of {‘disjoint,’ ‘equivalent’} has the effect of declaring that the constrainedElements are respectively disjoint or equivalent.

#### 9.1.2.2 Element (as extended)

**Package**: SEMOF
**isAbstract**: Yes
**Generalization**: Reflection::Object

**Description**

Element is extended with a new operation getMetaClasses to return multiple values. The original getMetaClass operation is retained. If there is only one metaclass, then getMetaClass will return it; otherwise, an exception will be thrown. Two additional operations provide reclassification capabilities. Note that the existing operation isInstanceOf can still be used to check whether an Element conforms to a class.

**Attributes**

No new attributes

### Associations

| | |
|---|---|
| /metaclass : UML::Classes::Kernel::Class [1..*]<br><br>(A_element_metaclass)( | A derived association providing navigation capabilities between metalevels. The association is navigable in both directions, but the association owns both ends. This association redefines the equivalent association defined by MOF Core, but with different multiplicity and navigation. |

### Operations

| | |
|---|---|
| getMetaClasses() : Class [1..*] | Returns the set of metaclasses that classify this element. |
| getMetaClass(): Class | Redefines MOF::Reflection::Element::getMetaClass(). If getMetaClasses only contains one class, this is returned by getMetaClass; otherwise, getMetaClass will throw an exception. |
| reclassify(oldMetaClass : Class [0..*],<br>newMetaClass: Class [0..*]) | This pair of operations provides the capability to reclassify any instance of SMOF::Element or its subclasses. Reclassification is not permitted for any element contained in package SMOF. |
| reclassifyAll(newMetaClass : Class [1..*]) | Reclassification of the element instance using either of the two operations is performed as an atomic step and results either in a complete reclassification, or has no effect at all. See “Semantics” below for the detailed description.<br><br>The operation reclassify() will throw an exception if oldMetaClass contains any metaclass that does not currently directly classify the object. |
| addMetaClass(newMetaClass : Class [1..*]) | Add the specified metaclasses to the classification of element. This is a convenience signature for reclassify() and equivalent to calling reclassify with an empty oldMetaClass argument. e.g.: reclassify( , new). |
| removeMetaClass(oldMetaClass : Class [1..*]) | Remove the specified metaclasses from the classification of element. This is a convenience signature for reclassify() and equivalent to calling reclassify with an empty newMetaClass argument. e.g.: reclassify(old, ). |
| container() : Element | Redefines MOF::Reflection::Element:container(). Returns the parent container of this element if any. Return Null if there is no containing element. If more than one container exists, which is possible in the case of multiple classification, a call to container will return Null and throw an exception. |
| getContainers() : Element [0..*] | Returns all existing parent containers for this element. |
| getContainerForMetaClass(metaClass : Class)<br>:Element | Returns the parent container, if any, defined by the classification by MetaClass. Returns Null if no such container exists. |

### Constraints

[1] Metaclasses to be added must not be abstract.
not self.getMetaClasses()->exists(isAbstract=true)

[2] Any element must be classified by at least one metaclass.
self.getMetaClasses()->size() >=1

[3] The metaclass association is derived from the getMetaClasses operation.
self.metaClass = self.getMetaClasses()

### Semantics

Any instance of SMOF::Element or its subclasses can be reclassified as constrained by the applicable Compatibility and Incompatibility elements.

Two operations, reclassify() and reclassifyAll() are provided to perform the reclassification (see below for the difference). Reclassification is performed as an atomic step: either the element instance is reclassified by the resulting set of classes derived during operation execution and all related side effects on all affected features of the element instance are completely performed, or the operation execution has no effect on the element instance at all and will signal its failure.

The signature of reclassify() has two input parameters: oldMetaClass lists the classes to be removed, newMetaClass lists the classes to be added to the set of classes classifying the element instance. The signature of reclassifyAll() has only the parameter newMetaClass and implies that all existing classes shall be removed. Besides this, both operations implement identical behavior.

- Reclassification preserves the identity of the reclassified element instance.
- When the operation completes, at least one class must classify the element instance, and none of the classes classifying the element instance may be abstract.
- If the set of classes to be removed contains classes identical to classes in the set of classes to be added, then these classes are not removed, the corresponding classes in the set of classes to be added are discarded, and all values for features defined by these classes remain untouched.
- If a class contained in the set of classes to be removed defined some features of the element instance, which are identically defined again by a class in the set of classes to be added, then the existing feature values are preserved unchanged. (For example when an old and a new metaclass share a common ancestor, or where an old and a new metaclass are ancestors of one another).

Any attempt to use reclassify(), reclassifyAll(), or addMetaClass() to simultaneously classify an element by metaclasses marked as disjoint, or which have any ancestors marked as disjoint, will cause the reclassification to fail and an exception will be thrown.

A new operation getMetaClasses(), has been introduced to return a list of all classes classifying the Element on which the operation is performed.

The existing operation getMetaClass(), as defined in MOF::Reflection, is redefined to return either the single metaclass if there is one, or to throw an exception.

Association A_element_metaclass redefines the equivalent unidirectional association defined by MOF Core. The association is derived using the SMOF operation getMetaclasses(). It can be used by OCL expressions to navigate between Elements and their metaclasses.

Multiply classified Elements may participate in multiple containments concurrently at any time provided that each containment is defined by a different classifying metaclass, and the containment semantic is restricted to the slice of the Element defined by that metaclass. A “slice of the Element” is defined as the structural and behavioral features of the

Element defined by the corresponding metaclass. At most one slot within a slice for the opposite of a composite property may have a value, where “slot within a slice” means that the slot has a defining feature owned by the corresponding metaclass or one of its ancestors.

If a container Element is deleted, which contained a multiple classified Element through a composite relationship, then the classifying metaclass that defined the part end of the composite relationship must be removed from the contained Element, effectively deleting the slice of that Element defined by that metaclass.

#### 9.1.2.3 Factory

Factory has not changed from CMOF. If an Element with multiple classifications needs to be constructed, a two-step process must be applied:

1. Create the Element with single classification using one of the CMOF Factory operations create() or createElement().
2. Add additional metaclasses using the SMOF Element::addMetaClass() operation.

# 10 Abstract Semantics

This clause describes the abstract semantics of SMOF. It uses essentially the same approach as the abstract semantics of CMOF but is reformulated here. The semantics of the SMOF reflective operations are described by the effect of corresponding operations on an abstract semantic domain model.

## 10.1 SMOF Semantic Domain Model

This specification does not model the semantics of Extents, which are unchanged from the *MOF* specification. The goal of this clause is to model the new semantics of Elements including the possibility of multiple classifications. This covers the concepts of multiply classified Elements, their Properties and values of those properties, including creation and destruction.

The SMOF semantic domain model is an extended version of a subset of UML 2.4.1 L1. The reused subset of UML contains the following non-abstract classes, all of their superclasses, and all properties, associations, constraints, and operations defined on these: Class, Association, Property, and OpaqueExpression. In what follows, this subset of UML is called CAPO. The non-abstract classes introduced by the semantic domain model are Instance, Link, Slot, LinkSlot, and InstanceValue.

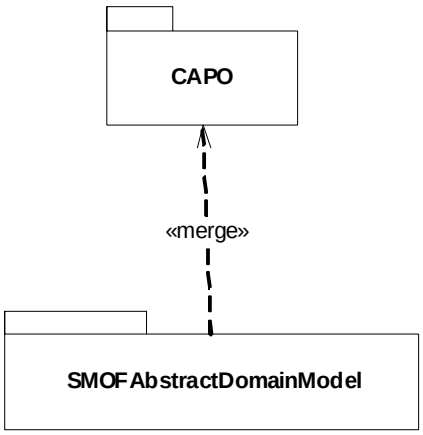


**Figure 10.1 - Semantic Domain Model Package**

The extensions are introduced to simplify the modeling of instances and links and to introduce new operations and constraints that define the semantics.

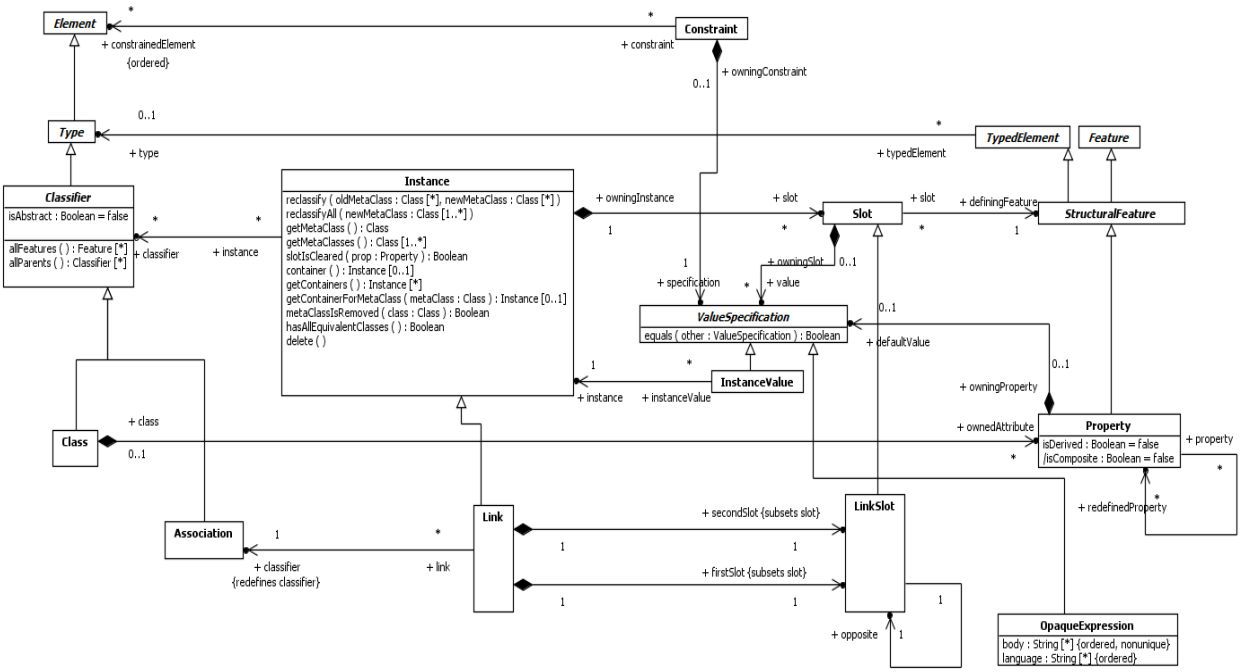


**Figure 10.2 - SMOFAbstractDomainModel package**

The semantics of SMOF::Element are modeled by instances of Instance according to the constraints and operations defined in what follows. To break any apparent circularity we assume that the semantics of instantiating the domain model itself are as defined in the *OCL 2* specification, which also of course allows us to use OCL to express constraints over instances of the abstract semantics domain model.

Slightly more formally, we are introducing a semantic function Φ that is a homomorphism from elements and operators in the *SMOF* specification to elements and operators in the semantic domain:

$\Phi : SMOF \rightarrow SMOF::AbstractDomainModel$

Such that for every n-ary operator μ:

$\Phi(\mu(a_1,.. ,a_n) = \Phi(\mu)(\Phi(a_1), ..., \Phi(a_n))$

Because CAPO shares its content with those aspects of UML that are merged into SMOF, much of Φ is simply an identity mapping. Hence Φ(SMOF::Class) = SMOFAbstractDomainModel::Class, Φ(SMOF::Property) = SMOFAbstractDomainModel::Property, and so on. Φ applied to any operation or attribute maps to a corresponding operation or attribute with the same name. Φ is the identity when applied to any data type or data value.

The interesting semantics are captured as follows.

For all instances obj of SMOF::Object:

-- Elements map to Instances
if (obj.isInstanceOfType(SMOF::Element, true)) then
Φ(obj).oclIsKindOf(SMOFAbstractDomainModel::Instance)

-- Links map to Links
if (obj.isInstanceOfType (SMOF::Link, true)) then
Φ(obj).oclIsKindOf(SMOFAbstractDomainModel::Link)

For all operations defined on classes in SMOF:

$\Phi(el.op(a_1,.. ,a_n)) = \Phi(el).\Phi(op)(\Phi(a_1), ..., \Phi(a_n))$

For all properties defined on classes in SMOF:

$\Phi(el.attr) = \Phi(el).\Phi(attr)$

Said in English, this means that the meaning of an operation or attribute applied to the element el is defined by the meaning of the corresponding operation or attribute in the semantic domain, with the mapping function applied to all of its arguments and results.

The following constraints and operations are introduced in the SMOFAbstractDomainModel package and apply to the classes in the merged semantic domain model in addition to all constraints defined in CAPO.

### 10.1.1 Class

#### Constraints

- NotBothEquivalentAndDisjoint
  No pair of classes exists such that they are both equivalent and disjoint

```
inv: not Class.allInstances()->exists(c | self.isEquivalentTo(c) and
self.isDisjointWith(c) or self.hasDisjointAncestorsWith(c)))
```

#### Operations

- isDisjointWith(other) :

```
post: result = Constraint.allInstances()->exists(c |
   c.specification.oclIsKindOf(OpaqueExpression)
   and c.specification.oclAsType(OpaqueExpression).language->at(0)='SMOF'
   and c.specification.oclAsType(OpaqueExpression)._'body'->at(0)='disjoint'
   and c.constrainedElement->includes(self)
   and c.constrainedElement->includes(other))
```

- isEquivalentTo(other) :

```
post: result = Constraint.allInstances()->exists(c |
   c.specification.oclIsKindOf(OpaqueExpression)
   and c.specification.oclAsType(OpaqueExpression).language->at(0)='SMOF'
   and c.specification.oclAsType(OpaqueExpression)._'body'->at(0)='equivalent'
```

```
    and c.constrainedElement->includes(self)
    and c.constrainedElement->includes(other))
```

- hasDisjointAncestorsWith(other) :

```
post: result = self.allParents()->exists(c1 | other.allParents()->exists( c2 |
  c1.oclAsType(Class).isDisjointWith(c2.oclAsType(Class))))
```

- directlyEquivalentClasses() :

```
post: result = Class.allInstances()->select(c | self.isEquivalentTo(c))
```

- thisAndAllParents() :

```
post: result = self->union(allParents()->collect(oclAsType(Class))->asSet())
```

### 10.1.2 Instance

#### Constraints

- OnlyClassesAndAssociations
  The classifiers can only be Classes or Associations.

```
inv: classifier->forAll(c | c.oclIsKindOf(Class) or c.oclIsKindOf(Association))
```

- LinksClassifiedByAssociations
  If the InstanceSpecification is not a Link, none of its classifiers are associations.

```
inv: not self.oclIsKindOf(Link) implies classifier->forAll(c |
  c.oclIsKindOf(Class))
```

- ClassifiersNotAbstract
  All classifiers are non-abstract.

```
inv: not classifier->exists(isAbstract)
```

- SlotsHaveDefiningProperties
  The defining feature of each slot is a structural feature (directly or inherited) of a classifier of the instance specification.

```
inv: slot->forAll(s | classifier->exists (c |
         c.allFeatures()->includes (s.definingFeature)))
```

- AtMostOneSlotPerFeature
  One structural feature (including the same feature inherited from multiple classifiers) is the defining feature of at most one slot in an instance specification.

```
inv: classifier->forAll(c | (c.allFeatures()->forAll(f |
               slot->select(s | s.definingFeature = f)->size() <= 1)))
```

- NoDisjointClasses
  No two metaclasses may be disjoint or have disjoint ancestors.

```
inv: let classes : Set(Class) = self.getMetaClasses() in
```

```
classes->forAll(c1 | not classes->exists(c2 | c1 <> c2 and
(c1.isDisjointWith(c2) or c1.hasDisjointAncestorsWith(c2))))
```

- AtLeastOneClassifier
  Each instance is classified at least once.

  ```
  inv: classifier->notEmpty()
  ```

- AllEquivalentClasses
  If any metaclasses or their ancestors have equivalent classes, then those equivalent classes are also classifiers, either directly or indirectly.

  ```
  inv: self.hasAllEquivalentClasses()
  ```

- AtMostOneContainerPerClassifier
  At most one slot within a slice for the opposite of a composite property may have a value.

  ```
  inv: let containerSlots : Set(Slot) = Link.allInstances()->select(link |
  link.secondSlot.value->any(true).oclAsType(InstanceValue).instance = self
  and
  link.secondSlot.definingFeature.oclAsType(Property).isComposite
  )->collect(firstSlot)->asSet() in
    classifier->forAll(cls |
      containerSlots->select(slot |
         cls.allFeatures()->includes(slot.definingFeature))->size() <= 1)
  ```

### Operations

- reclassify(oldMetaClassnewMetaClass) :

  ```
  pre: not newMetaClass->exists(isAbstract)

  pre: not self.classifier->exists(oclIsKindOf(Association))

  pre: let classesToRemove : Set(Class) = oldMetaClass->reject(o |
           newMetaClass->includes(o)) in
    let classesToAdd : Set(Class) = newMetaClass->reject(n |
           oldMetaClass->includes(n)) in
    let classesToLeave : Set(Class) = (self.getMetaClasses()->reject(c |
            classesToRemove->includes(c)))->union(classesToAdd) in
    classesToLeave->notEmpty() and classesToLeave ->forAll(ctl1 |
       not classesToLeave ->exists(ctl2 |
         ctl1.isDisjointWith(ctl2) or ctl1.hasDisjointAncestorsWith(ctl2)))

  pre: oldMetaClass->forAll(omc | getMetaClasses()->includes(omc))

  post: let classesToRemove : Set(Class) = oldMetaClass->reject(o |
            newMetaClass->includes(o)) in
    let classesToAdd : Set(Class) = newMetaClass->reject(n |
            oldMetaClass->includes(n)) in
  ```

```
  let classesToLeave : Set(Class) = (self.getMetaClasses()->reject(c |
          classesToRemove->includes(c)))->union(classesToAdd) in
  self.getMetaClasses()->includesAll(classesToLeave) and
 self.hasAllEquivalentClasses()

post: (slot@pre->reject(s | slot->includes(s)))->forAll(sl |
        self.slotIsCleared(sl.definingFeature.oclAsType(Property)))

post: (slot->reject(s | slot@pre->includes(s)))->forAll(sl |
      sl.value->any(true).equals(
          sl.definingFeature.oclAsType(Property).defaultValue))
```

- reclassifyAll(newMetaClass) :

```
pre: not newMetaClass->exists(isAbstract)

pre: not self.classifier->exists(oclIsKindOf(Association))

pre: newMetaClass ->forAll(nmc1 | not newMetaClass ->exists(nmc2 |
   nmc1.isDisjointWith(nmc2) or nmc1.hasDisjointAncestorsWith(nmc2)))

post:  self.getMetaClasses()->includesAll(newMetaClass) and
  self.hasAllEquivalentClasses()

post: (slot@pre->reject(s | slot->includes(s)))->forAll(sl |
       self.slotIsCleared(sl.definingFeature.oclAsType(Property)))

post: (slot->reject(s | slot@pre->includes(s)))->forAll(sl |
      sl.value->any(true).equals(
          sl.definingFeature.oclAsType(Property).defaultValue))
```

- getMetaClass() :

```
pre: self.getMetaClasses()->size() = 1
post: result = self.getMetaClasses()->one(true)
```

- getMetaClasses() :

```
post: result = self.classifier->select(oclIsKindOf(Class))->
       collect(oclAsType(Class))->asSet()
```

- slotIsCleared(prop) :

```
post: prop.isComposite and prop.type.oclIsKindOf(Class) implies
  Link.allInstances()->select(link |
  link.firstSlot.value->any(true).oclAsType(InstanceValue).instance = self
  and link.secondSlot.definingFeature.oclAsType(Property) = prop)
  ->collect(link |
         link.secondSlot.value.oclAsType(InstanceValue).instance)
  ->forAll(i | i.metaClassIsRemoved(prop.type.oclAsType(Class)))
```

- container() :

```
pre: self.getContainers()->size() <= 1
post: result = self.getContainers()->any(true)
```

- getContainers() :

```
post: result = Link.allInstances()->select(link |
   link.secondSlot.value->any(true).oclAsType(InstanceValue).instance = self
     and link.secondSlot.definingFeature.oclAsType(Property).isComposite)->
   collect(link |
     link.firstSlot.value->any(true).oclAsType(InstanceValue).instance)
   ->asSet()
```

- getContainerForMetaClass(metaClass) :

```
pre: Link.allInstances()->select(link |
  link.secondSlot.value->any(true).oclAsType(InstanceValue).instance = self and

      link.secondSlot.definingFeature.oclAsType(Property).isComposite and
      metaClass.thisAndAllParents()->
        includes(link.secondSlot.definingFeature.type.oclAsType(Class)))->
    collect(link |
       link.firstSlot.value->any(true).oclAsType(InstanceValue).instance)
       ->asSet()->size() <= 1

post: result = Link.allInstances()->select(link |
  link.secondSlot.value->any(true).oclAsType(InstanceValue).instance = self and

       link.secondSlot.definingFeature.oclAsType(Property).isComposite and
       metaClass.thisAndAllParents()->
          includes(link.secondSlot.definingFeature.type.oclAsType(Class)))->
     collect(link |
       link.firstSlot.value->any(true).oclAsType(InstanceValue).instance)
       ->asSet()->any(true)
```

- metaClassIsRemoved(class) :

```
post: (slot@pre->reject(s | slot->includes(s)))->forAll(sl |
   self.slotIsCleared(sl.definingFeature.oclAsType(Property)))

post: not self.getMetaClasses()->includes(class)
```

- hasAllEquivalentClasses() :

```
post: result = self.getMetaClasses()->forAll(c1 |
  let classAndAncestors : Set(Class) = c1.thisAndAllParents(),
     directEquivalences : Set(Class) = classAndAncestors->
       collect(directlyEquivalentClasses())->asSet() in
         directEquivalences->forAll(e |
           self.getMetaClasses()->exists(c2 |
             c2.thisAndAllParents()->includes(e)))
 )
```

- delete() :

```
post: slot@pre->forAll(sl |
   self.slotIsCleared(sl.definingFeature.oclAsType(Property)))
```

# 11 Changes to the XMI Serialization

Normally XMI element names are derived from the metamodel names: the root XMI element uses a metaclass name and the other elements use the name of the Property used to link the element to its container, with the xmi:type attribute indicating the actual metaclass (for single-inheritance metamodels xsi:type can be used).

xmi:ids only need to be unique within a document, and there is nothing to stop many xmi:ids being used for the same element in either the same, or different documents: they are all unified through using the same xmi:uuid. Though the use of xmi:uuid is generally optional in XMI, it is needed in such cases.

To allow the serialization of multiple classifications for an element, SMOF makes use of this existing mechanism with a separate XML element per class applied to a model element. Thus no changes are required to the *XMI* specification, and importers can deal with XMI documents from SMOF as they do with any other XMI document.

For example:

```
<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/20100901"
         xmlns:uml="http://www.omg.org/spec/UML/20100501"
         xmlns:bpmn="http://www.omg.org/spec/BPMN/20100501">
  <uml:Package name="P1" xmi:id="x1" xmi:type="uml:Package">
    <packagedElement xmi:id="x2" xmi:uuid="myorg.models.m555.e123” name="myClass"
       xmi:type="uml:Class">

             ... content related to uml:Class

    </packagedElement>
  </uml:Package>
  <bpmn:Definitions name = "Defs1" xmi:id="x3" xmi:type="bpmn:Definitions">
    <rootElements xmi:id="x4" xmi:uuid="myorg.models.m555.e123" name="myProcess"
       xmi:type=”bpmn:Process”>
             ...content related to bpmn:Process
    </rootElements >
  </bpmn:Definitions>
</xmi:XMI>
```

Note: in the above, the ‘name’ properties for the uml:Class and bpmn:Process are different.

Alternatively, the individual metaclass-related aspects could be serialized in different XMI files.

The above also represents one option for serializing multiple ownership (the same element having multiple composite owners through being multiple classified). Another option is to serialize the MOF Associations: this example uses a combination of XML nesting and an Association element in the same file; alternatively they could be in separate files with an href rather than an xmi:idref used:

```
<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/20100901"
         xmlns:uml="http://www.omg.org/spec/UML/20100501"
         xmlns:bpmn="http://www.omg.org/spec/BPMN/20100501">
  <uml:Package name="P1" xmi:id="x1" xmi:type="uml:Package">
    <packagedElement xmi:id="x2" xmi:uuid="myorg.models.m555.e123” name="myClass"
```

```
      xmi:type="uml:Class">

            ... content related to uml:Class

    </packagedElement>
  </uml:Package>
  <bpmn:Definitions name = "Defs1" xmi:id="x3" xmi:type="bpmn:Definitions"/>
  <bpmn:Process xmi:id="x4" xmi:uuid="myorg.models.m555.e123" name="myProcess"
       xmi:type=”bpmn:Process”>
            ...content related to bpmn:Process
  </bpmn:Process>
  <bpmn:A_definitions_rootElements>
    <definition xmi:idref="x3"/>
    <rootElements xmi:idref="x4"/>
  </bpmn:A_definitions_rootElements>
</xmi:XMI>
```

In the case where more than one metaclass shares the same property, the shared slots must be separately, and somewhat redundantly, serialized for each metaclass.

In order to provide control over how the metaclass aspects are serialized, additional options are added to the export option. Since this control over serialization is applicable to non-SMOF facilities, this represents a change to the general *MOF Facility and Object Lifecycle* specification.

In detail the change is as follows:

Add the following properties to the ExportOptions data type in section 6.10.3.2.1:

| | |
|---|---|
| onlyPackages:Package[0..*] | If a value is supplied for this property, only direct instances of classifiers in the specified packages are included in the export (in addition to those explicitly specified through *onlyClassifiers*). |
| onlyClassifiers: Classifier[0..*] | If a value is supplied for this property, only instances of the specified classifiers are included in the export, in addition to those specified through the *onlyPackages* property. Unlike that other property, specifying Classifiers in this property includes subtypes. |