Date: November 2010

MOF Support for Semantic Structures (SMOF)

FTF – Beta 1

OMG Document Number: ptc/2010-11-39

Standard Document URL: http://www.omg.org/spec/SMOF/1.0

Associated File(s)*: http://www.omg.org/spec/SMOF/20100801

* Original file: ad/2010-08-07.zip

This OMG document replaces the submission document (ad/2010-08-06, Alpha). It is an OMG adopted Beta specification and is currently in the finalization phase. Comments on the content of this document are welcome, and should be directed to issues@omg.org by August 2, 2011.

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 September 30, 2011. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

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/technology/agreement.)

Table of Contents

0 Preface

0.1 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. A Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM)

Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications

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

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
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.

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:

- SMOF for CMOF
- 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 2.0, OMG Document formal/06-01-01

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

OMG Unified Modeling Language™ (UML), Infrastructure, Version 2.3, OMG Document formal/2009-09-10

MOF 2.0/XMI Mapping, Version 2.1.1, OMG Document formal/2007-12-02

The Object Constraint Language (OCL) Version 2.2 is used to define constraints and semantics in subsequent clauses of this specification. The OCL 2.2 language definition can be found here:

Object Constraint Language Specification, Version 2.0, OMG Document formal/06-05-01

3.2 List of Non-Normative References

The following specifications are mentioned in descriptive text of subsequent clauses, but do not constitute a normative part of this specification:

OMG Unified Modeling Language™ (UML), Superstructure, Version 2.3, OMG Document ptc/2009-09-08

Semantics of a Foundational Subset for Executable UML Models, Version 1.0, OMG Document ptc/2010-02-03

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 defines a UML profile to enable an SMOF metamodel to be specified in standard UML. Clause 12 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 2.0
- MOF Facility Object Lifecycle 2.0

6.3 Acknowledgements

The following companies submitted this specification:

- 88solutions
- Adaptive
- Deere & Company
- Mega
- Microsoft
- Model Driven Solutions
- Sandpiper Software

The following companies supported this specification:

- Computer Science Corporation

7 Concept Overview and Use Cases

[Informative]

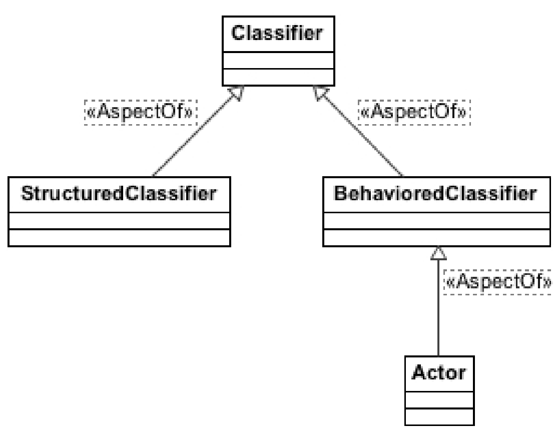
7.1 Overview

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 or their fundamental type, 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 Use Case: UML

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



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.

7.3 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 RFP is to allow SBVR to represent the types of the domain directly in MOF.

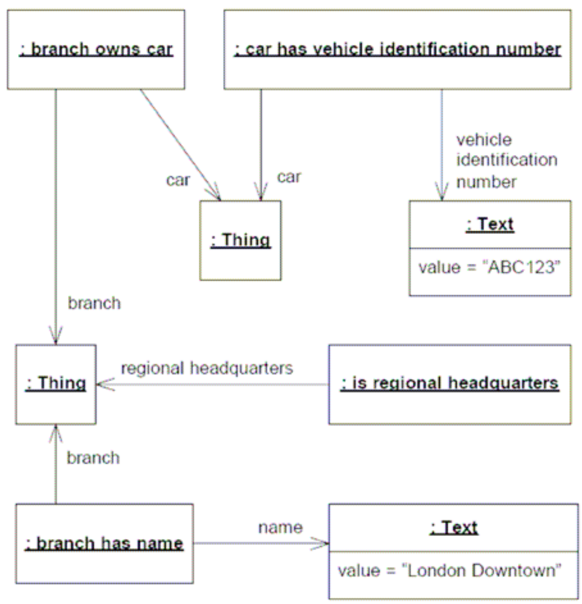
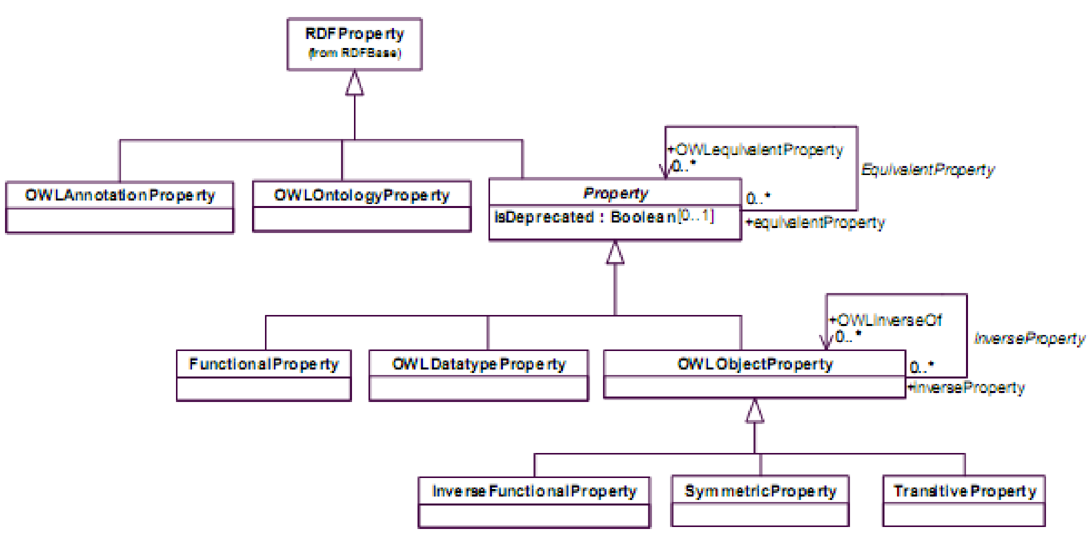


Figure L.2 - Instance diagram of facts expressed using EU-Rent English Vocabulary

7.4 Use Case: Ontology Definition Metamodel (ODM)

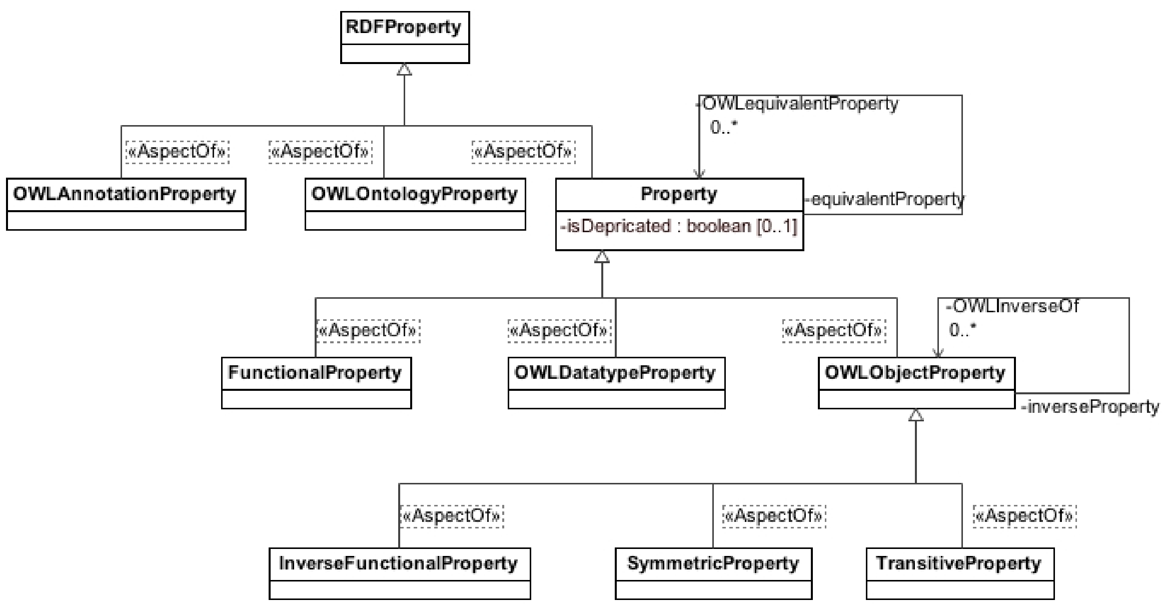
One of the incentives for the SMOF RFP was the requirement in OMG specifications for multiple classification. This issue was identified in SBVR as well as "ODM" (Ontology Definition Metamodel). ODM provides a MOF meta model of multiple ontology languages, including OWL. The following model fragment is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



Note that there are sever 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.



8 Abstract Syntax Architecture

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.

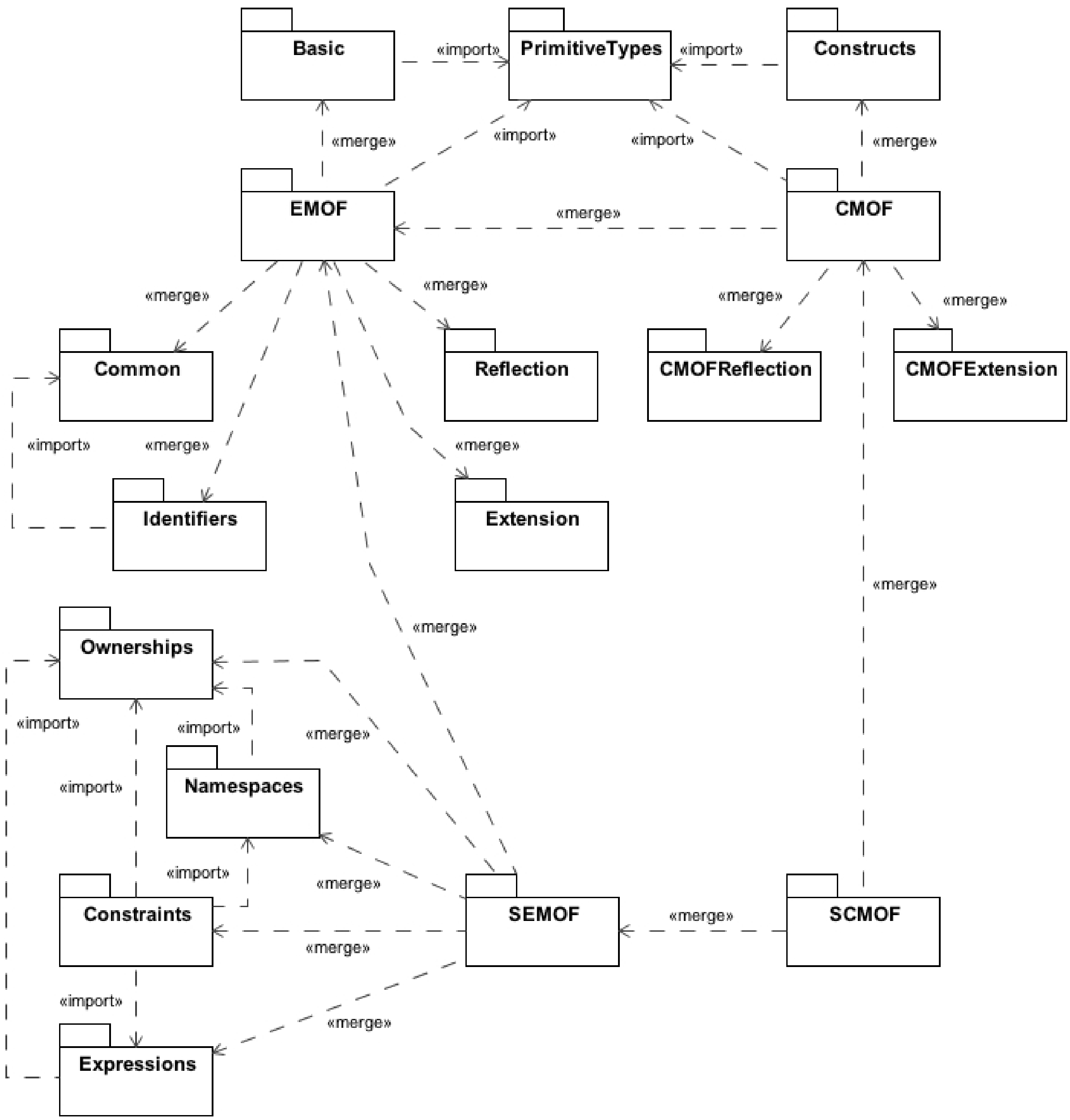


Figure 1 - The SMOF Packages in relation to the EMOF / CMOF Packages

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, with the exception of the new association between Element and its metaclasses since EMOF does not support associations. The SMOF extensions directly require Abstractions::Constraints and Abstractions::Expressions, which in turn require Abstractions::Ownership and Abstractions::Namespaces. These four packages are all part of Constructs, but are lacking from EMOF. Therefore package SEMOF merges these four InfrastructureLibrary packages explicitly.

Package SCMOF merges SEMOF with CMOF and introduces the new derived association between Element and Class to facilitate navigation to metaclasses.

9 Metamodel Extensions

9.1 Common SMOF Extensions

9.1.1 Abstract Syntax

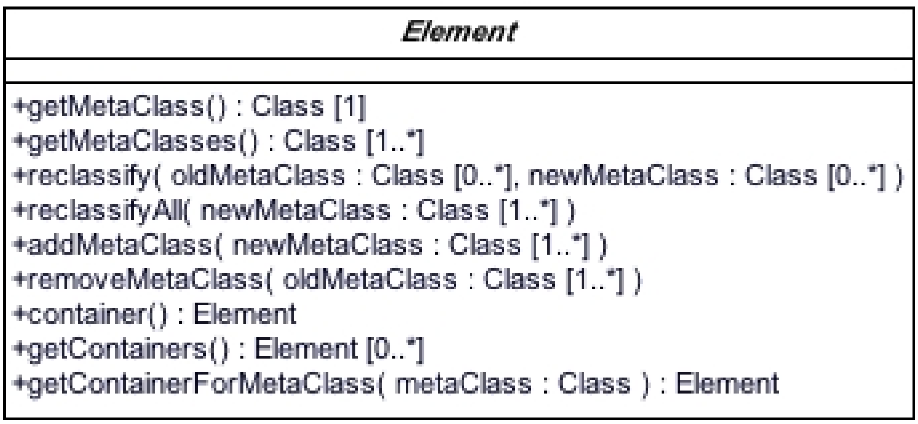


Figure 2 – Reflection, as extended by SMOF

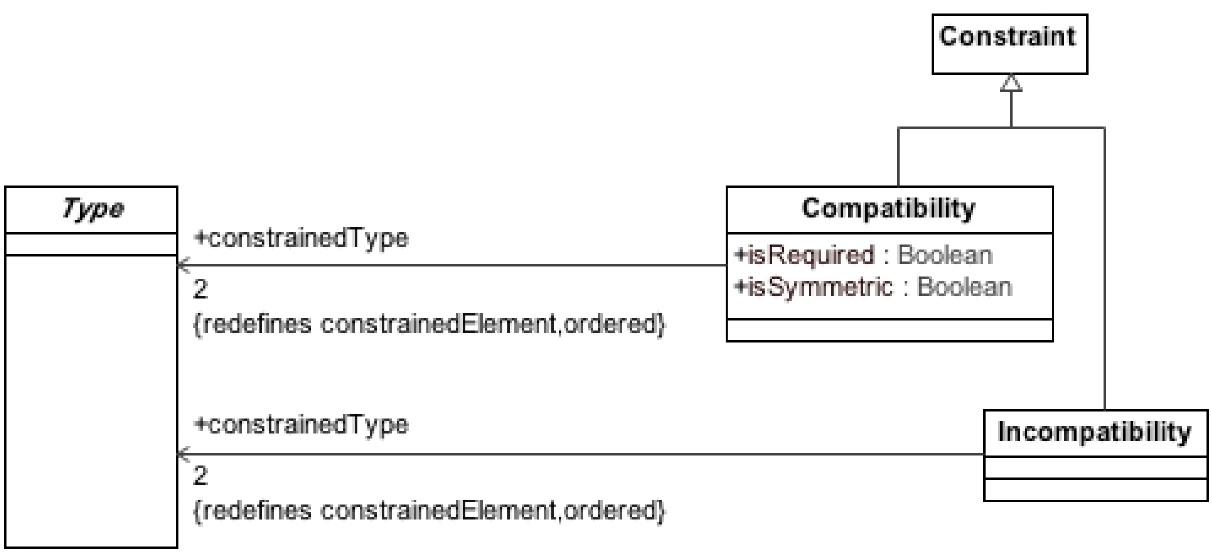


Figure 3 - SMOF Classification Constraints

9.1.2 Class Descriptions

9.1.2.1 Incompatibility

Package: SEMOF
isAbstract: No
Generalization: Abstractions::Constraints

Description

A subclass of Constraint, providing the ability to define an incompatibility rule between two potential types (and therefore also metaclasses).

Attributes

No new attributes

Associations

constrainedType : Type [2] {ordered}	Redefines constrainedElement inherited from Constraint to more precisely identify a pair of types declared as incompatible for concurrent participation in the classification of an element.

Operations

No new operations.

Constraints

No new constraints.

9.1.2.2 Compatibility

Package: SEMOF
isAbstract: No
Generalization: Abstractions::Constraint

Description

A subclass of Constraint, providing the ability to define a compatibility rule between two potential types (and therefore also metaclasses).

Attributes

isRequired : Boolean	If true, and if the constraint's specification evaluates to true, an instance of the target type will automatically be classified by the source type, where the source type is constrainedType->at(1) and the target type is constrainedType->at(2).
isSymmetric : Boolean	If true, the Compatibility constraint between the two referenced types becomes symmetric, which is equivalent to two identical Compatibility constraints in opposite direction applied to the two types.

Associations

constrainedType : Type [2] {ordered}	Redefines constrainedElement inherited from Constraint to more precisely identify a pair of types declared as compatible for concurrent participation in the classification of an element.

Operations

No new operations.

Semantics

Where an instance of Compatibility exists between two classes it is then permissible to classify an element by the source class as well as the target class, as long as the constraint's specification evaluates to true and there are no conflicting constraints (such as Incompatibility constraints). Where no instance of Compatibility exists (the default) it is not permissible to create such a multiple classification.

Where isRequired is true instances of the target classes are automatically additionally classified by the source class provided that the constraint's specification is true, and will be declassified if the constraint's specification becomes false.

If isRequired and isSymmetric are both true and the specification of the constraint evaluates to true, then the types are equivalent

9.1.2.3 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

No new associations.

Operations

getMetaClasses() : Class [1..*]	Returns the set of metaclasses which 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..*], 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 section "Semantics" below for the detailed description.
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) : 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

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)

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.

9.1.2.4 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.

9.2 SMOF Extensions for CMOF

9.2.1 Abstract Syntax

Figure 4 – Additional SMOF Extension for CMOF

9.2.2 Class Descriptions

9.2.2.1 Element (as extended)

Package: SCMOF
isAbstract: Yes
Generalization: Reflection::Object

Description

Package SCMOF provides a merge increment to Element, which adds the association A_element_metaClass. This association may be used in OCL expressions (or similar languages) to navigate to the Element's metaclasses.

Attributes

No new attributes

Associations

/metaClass : SCMOF::Class [1..*] (A_element_metaClass)	A derived association providing navigation capabilities between metalevels. The association is navigable in both directions, but the association owns both ends.

Operations

No new operations

Constraints

[1] The metaClass association is derived from the getMetaClasses operation.

```
self.metaClass = self.getMetaClasses()
```

9.2.2.2 Class (as extended)

Package: SCMOF
isAbstract: No
Generalization: Constructs::Classifier

Description

Package SCMOF provides a merge increment to Class, which adds the association A_element_metaClass. This association may be used in OCL expressions (or similar languages) to navigate to the Elements and their metaclasses.

Attributes

No new attributes

Associations

/element : SCMOF::Element [*] (A_element_metaClass)	A derived association providing navigation capabilities between metalevels. The association is navigable in both directions, but the association owns both ends.

Operations

No new operations

Constraints

No new constraints

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 the UML instance model constructed by merging in some additional elements and constraining the result.

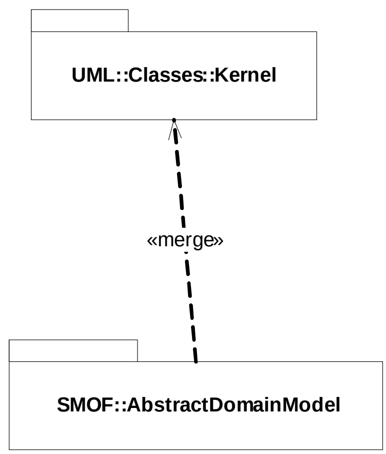


Figure 5 – Semantic Domain Model Package

The extensions are introduced to simplify the modeling of links (association instances), and to enable modeling of collection values and compatible and incompatible classifiers.

.

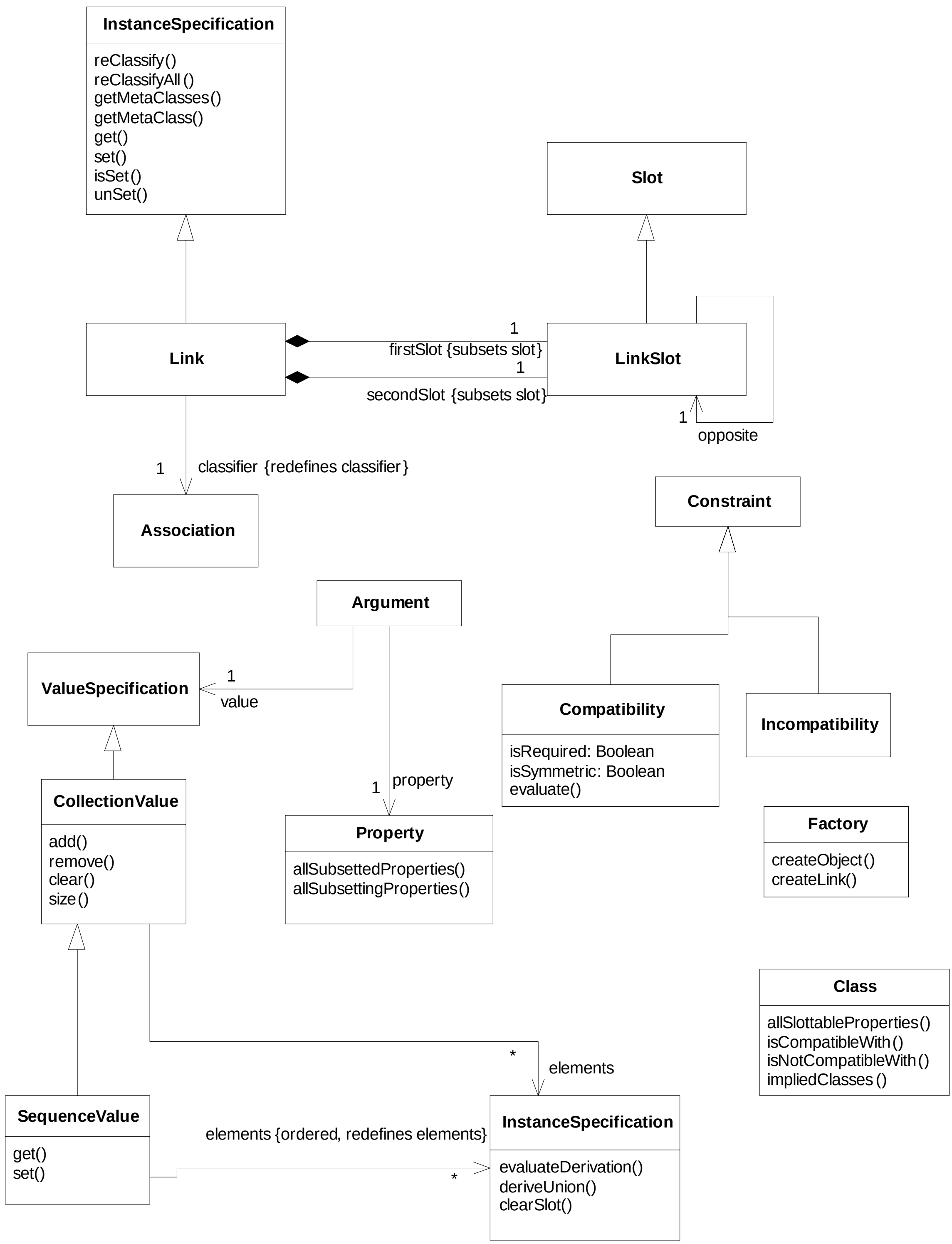


Figure 6 – AbstractDomainModel package

The semantics of SMOF::Element are modeled by instances of InstanceSpecification 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:

Φ : SMOF → 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 UML Kernel shares most of its content with those aspects of UML infrastructure that are merged into SMOF, much of Φ is simply an identity mapping. Hence Φ(SMOF::Class) = SMOF::AbstractDomainModel::Class, Φ(SMOF::Property) = SMOF::AbstractDomainModel::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 InstanceSpecifications
if (obj.isInstanceOfType(SMOF::Element, true)) then
    Φ(obj).oclIsKindOf(SMOF::AbstractDomainModel::InstanceSpecification)

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

-- ReflectiveCollections map to CollectionValues
if (obj.isInstanceOfType(SMOF::ReflectiveCollection)) then
    Φ(obj).oclIsKindOf(SMOF::AbstractDomainModel::CollectionValue)

-- ReflectiveSequences map to SequenceValues
if (obj.isInstanceOfType(SMOF::ReflectiveSequence)) then
    Φ(obj).oclIsKindOf(SMOF::AbstractDomainModel::SequenceValue)
```

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 AbstractDomainModel package and apply to the classes in the merged semantic domain model in addition to all constraints in UML Kernel.

10.1.1 InstanceSpecification

Constraints

The classifiers can only be Classes or Associations.

context: InstanceSpecification
inv:
classifier->forall(c | c.oclIsKindOf(Class) or c.oclIsKindOf(Association))

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

context: InstanceSpecification
inv:
not self.oclIsKindOf(Link) implies classifier->forall(c | c.oclIsKindOf(Class))

All classifiers are non-abstract.
context: InstanceSpecification
inv:
not classifier->exists(isAbstract)

There are no slots for derived or redefining properties.
context: InstanceSpecification
inv:
not slot->exists(s |
let p = s.definingFeature.oclAsType(Property) in p.isDerived or not p.redefinedProperty->isEmpty())

The defining feature of each slot is a structural feature (directly or inherited) of a classifier of the instance specification.
context: InstanceSpecification
inv:
slot->forAll(s | classifier->exists (c | c.allFeatures()->includes (s.definingFeature)))

One structural feature (including the same feature inherited from multiple classifiers) is the defining feature of at most one slot in an instance specification.
context: InstanceSpecification
inv:
classifier->forAll(c | (c.allFeatures()->forAll(f | slot->select(s | s.definingFeature = f)->size() <= 1)))

No two classifiers may be related by an Incompatibility.
context: InstanceSpecification
inv:
classifier->forAll(c1 | not classifier->exists(c2 | c1 <> c2 and c1.isNotCompatibleWith(c2))

Every classifier must be related by Compatibility to another classifier.
context: InstanceSpecification
inv:
classifier->forAll(c1 | classifier->forall(c2 | c1 = c2 or c1.isCompatibleWith(c2))

If any classifiers are implied, they are present.
context: InstanceSpecification
inv:
classifier->forAll(c1 | c1.impliedClasses()->forall(c2 | classifier->includes(c2))

Operations

container() : InstanceSpecification [0..1]
pre:
self.getContainers()->size() <= 1
post:
result = self.getContainers()->any(true)

getContainers() : InstanceSpecification [0..*]
post:
```
result = Link.allInstances()->select(link |
      link.secondSlot.value.oclAsType(InstanceValue).instance = self
    and
      link.secondSlot.definingFeature.oclAsType(Property).isComposite)->collect(link |
            link.firstSlot. value.oclAsType(InstanceValue).instance)
```

getContainerForMetaClass(metaClass: Class) : InstanceSpecification [0..1]
pre:
```
Link.allInstances()->select(link |
      link.secondSlot.value.oclAsType(InstanceValue).instance = self
    and
      link.secondSlot.definingFeature.oclAsType(Property).isComposite
    and
      metaClass.allParents()->including(metaClass)->includes(link.secondSlot.definingFeature.type)
    )->collect(link |
            link.firstSlot. value.oclAsType(InstanceValue).instance).asSet()->size() <= 1
```

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

getMetaClasses() : Class [1..*] { ordered }
post:
result = self.classifier

getMetaClass() : Class
pre:
self.classifier->size() = 1
post:
result = self.classifier->one(true)

reclassify(oldMetaClass : Class[0..*], newMetaClass : Class[0..*])
pre:
not newMetaClass->exists(isAbstract)
pre:
not self.classifier->exists(oclIsKindOf(Association)) and
not newMetaClass->exists(oclIsKindOf(Association))
pre:

let classesToRemove = oldMetaClass – newMetaClass in
let classesToAdd = newMetaClass – oldMetaClass in
let classesToLeave = (classifier – classesToRemove)->union(classesToAdd) in
classesToLeave->size() > 0
and classesToLeave->forall(ctl1 | not classesToLeave->exists(ctl2 |
ctl1 <> ctl2 and ctl1.isNotCompatibleWith(ctl2)))
and classesToAdd->forall(addedClass |
classesToLeave->exists(existingClass |
addedClass <> existingClass and
addedClass.isCompatibleWith(existingClass)))

post:
let classesToRemove = oldMetaClass – newMetaClass in
let classesToAdd = newMetaClass – oldMetaClass in
let classesToLeave = (classifier – classesToRemove)->union(classesToAdd) in
classifier = classesToLeave->collect(ctl | ctl->impliedClasses())

post:
(slot@pre – slot)->forall(sl | self.clearSlot(sl.definingFeature.oclAsType(Property)))

post:
(slot – slot@pre)->forall(sl | sl.value = sl.definingFeature.oclAsType(Property).defaultValue)

reclassifyAll(newMetaClass : Class[1..*])

pre:
not newMetaClass->exists(isAbstract)

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

pre:
newMetaClass ->forall(nmc1 | not newMetaClass ->exists(nmc2 |
nmc1 <> nmc2 and nmc1.isNotCompatibleWith(nmc2)))
and newMetaClass ->forall(addedClass |
newMetaClass->exists(existingClass |
addedClass <> existingClass and
addedClass.isCompatibleWith(existingClass)))

post:
classifier = newMetaClass->collect(ctl | ctl->impliedClasses())

post:
(slot@pre – slot)->forall(sl | self.clearSlot(sl.definingFeature.oclAsType(Property)))

post:
(slot – slot@pre)->forall(sl | sl.value = sl.definingFeature.oclAsType(Property).defaultValue)

get(prop: Property) : ValueSpecification

pre:
classifier.collect(ownedAttribute).asSet().includes(prop)

pre:
-- if a property redefines several other properties they all have the same value
not prop.redefinedProperty->isEmpty() implies prop.redefinedProperty->forall(red1 |
prop.redefinedProperty->forall(red2 | self.get(red1) = self.get(red2)))

post:
-- specify the type of the result
prop.upper <> 1 implies result.oclIsKindOf(CollectionValue)
and

prop.upper <> 1 and prop.isOrdered implies result.oclIsKindOf(SequenceValue)
and
prop.upper = 1 and prop.type.oclIsKindOf(Class) implies result.oclIsKindOf(InstanceValue)
post:
-- non-derived attributes
self.slot->exists(definingFeature = prop) implies
let v = self.slot->any(definingFeature = prop).value in
if v->isEmpty() then result = prop.defaultValue else result = v
post:
-- derived properties
prop.isDerived and not prop.isDerivedUnion implies result = evaluateDerivation(prop)
post:
-- derived unions
prop.isDerivedUnion implies result = self->deriveUnion(prop)
post:
-- redefining properties
not prop.redefinedProperty->isEmpty() implies result = self.get(prop.redefinedProperty->any(true))

set(prop: Property, value: ValueSpecification)
pre:
classifier.collect(ownedAttribute).asSet().includes(prop)
pre:
not prop.isDerived and not prop.isReadOnly
pre:
-- if a property redefines several other properties they all have the same value
not prop.redefinedProperty->isEmpty() implies prop.redefinedProperty->forall(red1 |
prop.redefinedProperty->forall(red2 | self.get(red1) = self.get(red2)))

post:
-- non-derived attributes
self.slot->exists(definingFeature = prop) implies self.slot->any(definingFeature = prop).value = value
post:
-- redefined properties
not prop.redefinedProperty->isEmpty() implies prop.redefinedProperty.forall(red **|** self.set(red, value))

isSet(prop: Property) : Boolean
pre:
classifier.collect(ownedAttribute).asSet().includes(prop)
post:
result = (not self.slot->any(definingFeature = prop).value->isEmpty())

unSet(prop: Property)
pre:
classifier.collect(ownedAttribute).asSet().includes(prop)
post:
self.slot->any(definingFeature = prop).value= prop.defaultValue

delete()
post:
self.slot->forall(sl | self.clearSlot(sl.definingFeature.oclAsType(Property)))

clearSlot(Property prop)
post:

```
  if prop.isComposite then
    link.allInstances()->select(link |
       link.firstSlot.value.oclAsType(InstanceValue).instance = self
     and
       link.secondSlot.definingFeature.oclAsType(Property).isComposite)->collect(link |
             link.secondSlot.value.oclAsType(InstanceValue).instance)->forall(delete())
```

```
deriveUnion(Property prop) : ValueSpecification
  pre:
    prop.isDerivedUnion and prop.definingFeature.type.oclIsKindOf(Class)
  post:
    let linksSourcedOnSelf = Link.allInstances()->select(link |
        link.firstSlot.value.oclAsType(InstanceValue).instance = self)
    in let linksTargetedOnSelf = Link.allInstances()->select(link |
        link.secondSlot.value.oclAsType(InstanceValue).instance = self)
    in let subsettingLinksSourcedOnSelf = linksSourcedOnSelf->select(link |
          self.allSubsettingProperties->includes(link.secondSlot.definingFeature.oclAsType(Property)))
    in let subsettingLinksTargetedOnSelf = linksTargetedOnSelf ->select(link |
          self.allSubsettingProperties->includes(link.firstSlot.definingFeature.oclAsType(Property)))
    in let allTargets = subsettingLinksSourcedOnSelf->collect(link |
          link.secondSlot.value.oclAsType(InstanceValue).instance)
    in let allSources = subsettingLinksTargetedOnSelf ->collect(link |
          link.firstSlot.value.oclAsType(InstanceValue).instance)
    in let allElements = allTargets ->union(allSources)
    in
      if allElements->size() = 1
      then
        result.oclIsKindOf(InstanceValue) and result = allElements->one(true)
      else
        result.oclIsKindOf(CollectionValue) and result.oclIAsType(CollectionValue).elements = allElements
```

```
evaluateDerivation(Property prop) : ValueSpecification
  -- return the result of evaluating the derivation expression according to the semantics of its language
```

10.1.2 Constraint

Constraints

None additional.

Operations

10.1.3 Class

Constraints

None additional.

Operations

```
allSlottableProperties() : Property [0..*]
post:
   result = self.ownedAttribute->select( prop |
     not prop.isDerived and prop.redefinedProperty->isEmpty()
     )->union(superclass->collect(allSlottableProperties())
```

```
isCompatibleWith(other : Class)
  pre:
    self <> other
  post:
    result =
        Compatibility.allInstances()->exists(cmp I cmp.check(self, other))
      or
        self.allParents()->includes(other)
      or
        other.allParents()->includes(self)

isNotCompatibleWith(other : Class)
  pre:
    self <> other
  post:
    result =
      Incompatibility.allInstances()->exists(inc | inc.check(self, other))

impliedClasses() : Class [1..*]
  post:
    result = Set{self} ->union(Compatibility.allInstances()->select(cmp |
                 cmp.isRequired and
                   (cmp.constrainedElement->at(1) = self or
                      (cmp.isSymmetric and cmp.constrainedElement->at(2) = self)))->collect( cmp|
                          cmp.constrainedElement->collect(el | oclAsType(Class))))
```

10.1.4 Property

Constraints

Derived unions are only defined for properties whose type is a class.

```
context: Property
inv:
  isDerivedUnion implies type.oclIsKindOf(Class)
```

Operations

```
allSubsettedProperties() : Property[0..*]
pre:
    self.type.oclIsKindOf(Class)
post:
    result = Property.allInstances()->select(prop |
        self.subsettedProperty->includes(prop) or
        self.subsettedProperty->collect(sub | sub.allSubsettedProperties->includes(prop))

allSubsettingProperties() : Property[0..*]
pre:
    self.type.oclIsKindOf(Class)
post:
    result = Property.allInstances()->select(prop | prop.allSubsettedProperties->includes(self))
```

10.1.5 Link

Constraints

There is only one classifier and it is an association.
context: Link
inv:
```
classifier->size() = 1 and classifier->one(true).oclIsKindOf(Association)
```

If a Link represents a composition, then secondSlot.definingFeature.isComposite is true
context: Link
inv:
```
not firstSlot.definingFeature.oclAsType(Property).isComposite
```

The link slots are opposites
context: Link
inv:
```
firstSlot.opposite = secondSlot and secondSlot.opposite = firstSlot
```

Operations

equals(otherLink : Link) : Boolean
post:
```
  result = (self.association = otherLink.association and
            self.firstSlot.value.oclAsType(InstanceValue).instance =
                      otherLink.firstSlot.value.oclAsType(InstanceValue).instance and
            self.secondSlot.value.oclAsType(InstanceValue).instance =
                      otherLink.secondSlot.value.oclAsType(InstanceValue).instance)
```

10.1.6 LinkSlot

Constraints

The value must evaluate to an element.
context: LinkSlot
inv:
```
value.oclIsKindOf(InstanceValue)
```

Where the property is navigable, the instance slot is compatible with the link slot (i.e. look in the element found in the opposite slot; if it has a slot with the same property then the value must be the same).
context: LinkSlot
inv:
```
let oppositeElement = opposite.value.oclAsType(InstanceValue).instance in
let property = definingFeature.oclAsType(Property) in
let oppositeElementSlot = oppositeElement.slot->any(sl | sl.definingFeature = property) in
  not oppositeElementSlot ->isEmpty() implies
    oppositeSlot.value. oclAsType(InstanceValue).instance = value.oclAsType(InstanceValue).instance
```

Operations

None.

10.1.7 Slot

Constraints

The value is compatible with the multiplicity and type of the defining property

context: Slot
inv:

```
let prop = definingFeature.oclAsType(Property) in
    prop.upper <> 1 implies value.oclIsKindOf(CollectionValue)
  and
    prop.upper <> 1 and prop.isOrdered implies value.oclIsKindOf(SequenceValue)
  and
    prop.upper = 1 and prop.type.oclIsKindOf(Class) implies value.oclIsKindOf(InstanceValue)
```

Operations

None.

10.1.8 Incompatibility

Constraints

The constrainedElement collection contains two different elements.

context: Incompatibility
inv:

```
self.constrainedElement->size() = 2
  and
constrainedElement->at(1) <> constrainedElement->at(2)
```

The specification is a LiteralBoolean with value true.

context: Incompatibility
inv:

```
self.specification.oclIsKindOf(LiteralBoolean)
  and
self.specification.oclAsType(LiteralBoolean).value = true
```

Operations

check(first : Class, second: Class) : Boolean
pre:
first <> second
post:
result = constrainedElement->includes(first) and constrainedElement->includes(second)

10.1.9 Compatibility

Constraints

The constrainedElement collection contains two different elements.

context: Compatibility
inv:

```
self.constrainedElement->size() = 2
   and
```

  constrainedElement->at(1) <> constrainedElement->at(2)

Operations

evaluate() : Boolean
-- evaluate the specification as a Boolean expression in the context of the first constrained element

check(first : Class, second: Class) : Boolean
 pre:
  first <> second
 post:
  result = self.evaluate() and
   ((constrainedElement->at(1) = first and constrainedElement ->at(2) = second) or
   (self.isSymmetric and
    constrainedElement ->at(2) = first and constrainedElement ->at(1) = second))

10.1.10 Factory

Constraints

None.

Operations

createElement(class: Class, arguments : Argument[0..*]) : InstanceSpecification
pre:
  arguments->forall(value.oclIsKindOf(LiteralSpecification))
pre:
  arguments->forall(arg | class.member->includes(arg.property))

post:
  class.allSlottableProperties->forall(prop | result.slot->one(definingFeature = prop))
post:
 arguments->forall(arg | result.slot->one(definingFeature = arg.property).value = arg.value)
post:
  let argInitializedSlots =
   arguments->collect(arg | result.slot->one(definingFeature = arg.property))
  in
   (result.slot – argInitializedSlots)->forall(slot |
    slot.value = slot.definingFeature.oclAsType(Property).defaultValue)

createLink(ass : Association, first : InstanceSpecification, second : instanceSpecification) : Link
pre:
  (first.classifier->includes(ass.memberEnd->at(1).type)
 and
  second.classifier->includes(ass.memberEnd->at(2).type))
or
 (first.classifier->includes(ass.memberEnd->at(2).type)
 and
  second.classifier->includes(ass.memberEnd->at(1).type))

post:

```
  result.firstSlot.oclAsType(InstanceValue).instance = first
and
  result.secondSlot.oclAsType(InstanceValue).instance = second
or
  result.firstSlot.oclAsType(InstanceValue).instance = second
and
  result.secondSlot.oclAsType(InstanceValue).instance = first
```

post:

```
 ass.memberEnd.any(isComposite) implies
          result.secondSlot.definingFeature.oclAsType(Property).isComposite
```

11 Semantic MOF Profile

11.1 Overview

The following UML profile elements are provided to enable a SMOF meta model to be specified in standard UML. The essential features of this profile are to manage when a MOF instance may be, must be or may not be classified by any two classifiers.

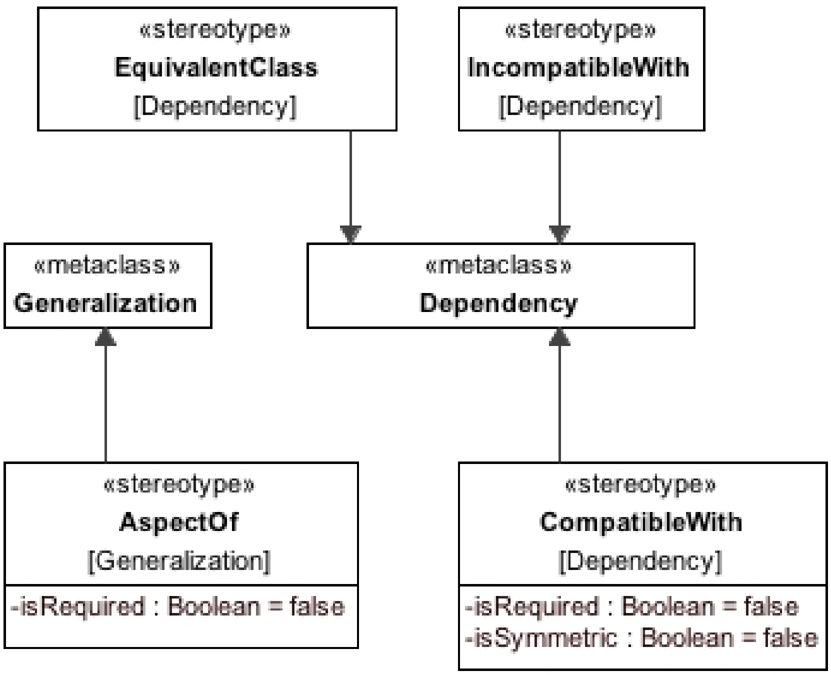


Figure 7 - SMOF Profile

11.2 Stereotype Descriptions

11.2.1 AspectOf

Package:
Stereotype of: Generalization

Description

It is common within a model to have a type of instance that may be categorized by any combination of subclasses and these subclasses may change over time. The additional classes represent aspects of the instance that may be added or removed during the life-cycle of the object. These additional classes, or aspects, of an object may be combined in arbitrary ways, except as may be prevented by a constraint or “IncompatibleWith” dependency.

Where AspectOf exists between two classes it is then permissible to add or remove the subtype during the lifetime of an instance. Alternatively, if AspectOf does not exist between two classes the subtype can not be added or removed from an instance. This represents a more conservative default than, say, OWL which allows any resource to be classified by any class unless otherwise constrained.

"AspectOf" is a stereotype of generalization which specifies that the aspect is a subtype of the target class and that the subtype may be added or removed at runtime. The Generalization with the AspectOf stereotype may have a Constraint (using normal UML modeling) to limit when the aspect may be applied.

Applying «AspectOf» to a Generalization from A to B is exactly equivalent to, and a shorthand for, creating a «CompatibleWith» Dependency with A as client and B as supplier and isSymmetric='false'.

When the subtype/aspect is added, the element remains directly classified by the superclass as well as the subclass. So, for example, oclIsKindOf() will be true for both the superclass and the subclass.

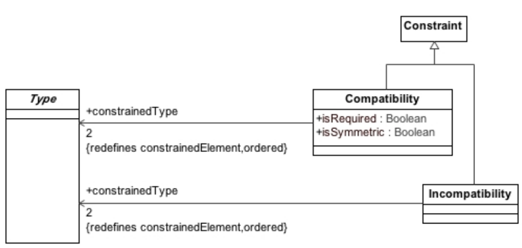
A Generalization that is not stereotyped as an aspect uses the more common "object oriented programming" semantics where an object must be created with a single type that can not be changed. Generalization with «AspectOf» applied (or linked by a «CompatibleWith» Dependency) corresponds more closely to the semantics of RDFS and OWL in that whatever is being modeled may be classified by any number of aspects, each with its own class.

Note that in MOF-2 subtypes are assumed to be non-overlapping (like Java or C#). Aspects are required to specify when the broader concept of generalization applies – that the same modeled individual may be classified in multiple ways. Base UML has the broader interpretation of generalization: AspectOf make the distinction specific.

Attributes

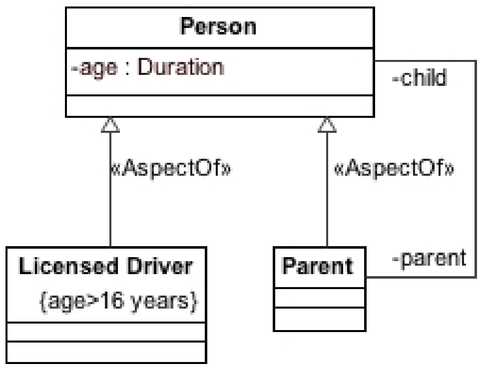
isRequired : Boolean = false	isRequired causes instances of the superclass to be automatically classified by the subclass provided any constraints on the superclass relation are true. If isRequired is false instances are allowed to, but not required to, add the subclass using the reclassify operations on the instance. Where isRequired is true, instances of the superclass that comply with the constraint (if any) will implicitly be classified with the subclass and declassified when the constraint (if any) becomes false. If there is a constraint the set of instances of the subclass will be that subset of the superclass set of instances where the constraint holds true. Where the aspect is required with no constraints, all instances of the superclass will be instances of the subclass.

SMOF Metamodel Effect



In addition to creating the "superClass" relation as normal, a Compatibility constraint is created, owned by the subtype. The ConstrainedType property is set with the subclass as the first element and the superclass as the second element. The isRequired property of the Compatibility constraint is set to the corresponding property value of the AspectOf stereotype

Example



The above model demonstrates the use of AspectOf applied to generalization. A person may be a "Licensed" Driver" and/or a "Parent". Within the lifetime of a Person they may become a parent or a licensed driver and may be both at the same time. All of the features and constraints of person apply to both licensed drivers and parents. Since they are subtypes, the age and any other parent properties are visible in both "Parent" and "Licensed Driver".

11.2.2 CompatibleWith

Package:
Stereotype of: Dependency

Description

It is common within a model to have a type of instance that may be categorized by any combination of other classes. The additional classes of an instance may be added or removed during the life-cycle of the object. These additional classes of an object may be combined in arbitrary ways, except as may be prevented by a constraint or "IncompatibleWith" statement.

"CompatibleWith" is a stereotype of Dependency and specifies that an instance may be classified by both classifiers and that the classifiers may be added or removed at runtime. The CompatibleWith dependency may have a constraint to limit when the compatibility holds.

Where CompatibleWith exists between two classes it is then permissible to add or remove the clientt classifier during the lifetime of an instance of the supplier classifier – that is you can add the subject classifier. Alternatively, if AspectOf or CompatibleWith does not exist between two classes (or their supertypes) an instance may not be explicitly classified by both classes (A class always implicitly classifies an instance by all superclasses of such a class). This represents a more conservative default than, say, OWL which allows any resource to be classified by any class unless otherwise constrained.

Attributes

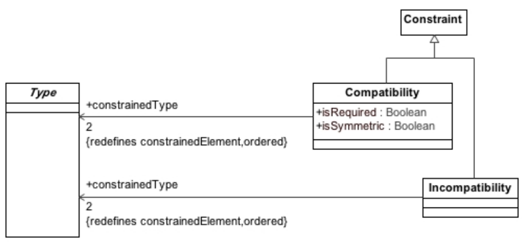
isRequired : Boolean = false	isRequired causes instances of the client class to be automatically classified by the supplier class provided any constraints on the isCompatible dependency are true. If isRequired is false instances are allowed to, but not required to, become instances of the client class using the reclassify operations on the instance.

isSymmetric : Boolean = false	isSymmetric is equivalent to having two CompatibleWith Dependencies where the inverse isCompatibleWith dependency is implied. The constraint, if any, is evaluated in the context of the first constrained element, but applies in both directions.

Constraints

[1] The source and target of the dependency must be types.

SMOF Metamodel Effect



A Compatibility constraint is created. Owned by the client class The constrainedType relation is set with the client class as the first element and the supplier class as the second element. The isRequired and isSymmetric properties of the Compatibility constraint are set to the corresponding property values of the CompatibleWith stereotype.

Example

The “Record club member” class may be added to a person – the classes are compatible as classifiers of any one instance.

11.2.3 IncompatibleWith

Package:
Stereotype of: Dependency

Description

“IncompatibleWith” specifies that two classes may not classify the same instance. Any attempt to have an instance classified by both results in an exception.

With so many options to multiply classify model elements based on aspects it is frequently required to prevent various combinations. IncompatibleWith specifies these illegal combinations.

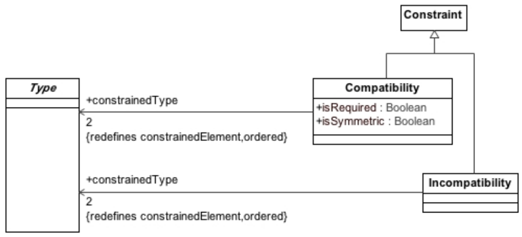
Attributes

none

Constraints

[1] The client and supplier of the dependency must be types.

SMOF Metamodel Effect



An Incompatibility constraint is created, owned by the client class. The constrainedType property is set with the client class as the first element and the supplier class as the second element.

Example

The above model fragment states that a Person can't be a Beverage.

11.2.4 EquivalentClass

Package:
Stereotype of: Dependency

Description

An «EquivalentClass» Dependency asserts that two classes have the same set of instances – that is that every instance of one class is an instance of the other. Equivalent class is frequently used when there are multiple names or representations of the same set of things.

This is shorthand for, and exactly equivalent, to applying the «CompatibleWith» stereotype to the same Dependency and setting isRequired and isSymmetric to true.

EquivalentClass in SMOF has the same intent as equivalentClass in OWL.

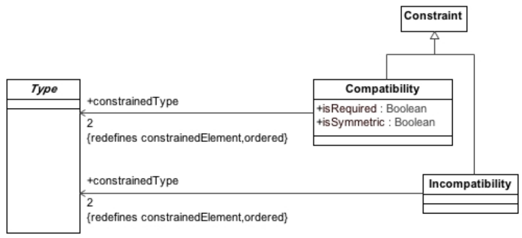
Attributes

none

Constraints

[1] The client and supplier of the dependency must be types.

SMOF Metamodel Effect



A Compatibility constraint is created owned by the client class. The constrainedType relation is set with the client class as the first element and the supplier class as the second element. The isRequired & isSymmetric properties as well as the specification of the Compatibility constraint are set to true.

Example

The above model states that people and humans are the same thing – all people are humans and all humans are people. It does not, however, merge these classes.

12 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/20100101"
         xmlns:uml="http://www.omg.org/spec/XMI/20100101"
         xmlns:bpmn="http://www.omg.org/spec/BPMN/20100101">
  <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 a href rather than an xmi:idref used:

```
<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/20100101"
         xmlns:uml="http://www.omg.org/spec/XMI/20100101"
         xmlns:bpmn="http://www.omg.org/spec/BPMN/20100101">
  <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.