Date: January 2006

Meta Object Facility (MOF) IDL Language Mapping Specification

OMG Available Specification
Version 2.0

formal/06-01-02

OBJECT MANAGEMENT GROUP

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

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.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. A catalog of all OMG 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. (as of January 16, 2006) at:

OMG Headquarters
160 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.

Issues

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

1 Scope

This specification outlines a new MOF 2 IDL mapping. It defines mapping rules for elements of a MOF 2.0 compliant model to the CORBA Component Model (CCM). The mapping rules basically depend on the state of the MOF2.0 Core documents [6] and the adopted UML Infrastructure [9]. The set of mapping rules is outlined in Chapter 5 and defined in detail in Chapter 6.

We reuse the concepts of CCM as much as possible aiming at the generation of highly performant, highly scalable, and reliable repositories, which are automatically deployable. The authors believe that this specification benefits from the gained experiences during several years in developing and using metadata repositories and CORBA technology. Furthermore, the technical disadvantages of the current, rudimentary IDL break the need for an efficient and convenient access in every day usage of modeldatabases, which are at the very heart of the MDA.

For the definition of the mapping rules themselves, we derive one combined model from the MOF 2.0 model and the CCM metamodel. Based on this combined model, we define the mapping rules using OCL. The methodology is covered by Chapter 5, including assumptions of the MOF model that served as the basis for this specification.

2 Conformance

An implementation that conforms to the MOF 2 IDL specification must conform to one of these compliance points in MOF 2 Core (ptc/03-10-04) and must produce IDL for at least one of the following:

- EMOF-to-CCM: EMOF compliant models mapped using the CCM mapping rules defined in section 8.2 and 8.3 (except 8.2.11 and 8.2.8, Rule (17) only for EMOF Properties).
- EMOF-to-Base-IDL: EMOF compliant models mapped using the Base-IDL mapping rules defined in section 8.2 and 8.4 (except 8.2.11 and 8.2.8, Rule (17) only for EMOF Properties).
- CMOF-to-CCM: CMOF compliant models mapped using the CCM mapping rules defined in section 8.2 and 8.3.
- CMOF-to-Base-IDL: CMOF compliant models mapped using the Base-IDL rules defined in section 8.2 and 8.4.

Furthermore, additional division of each of these four main compliance points results from the aspect of Runtime configurability - compliant products must implement either

1. full runtime configuration facilities for the reference/value semantic or
2. only default (i.e., reference) semantic.

This means that the derived IDL is not altered, only the computational behavior changes (cp. section 9.1 and 9.3 "set_call_semantic" and "set_value_depth" operations).are specific.

3 Normative References

Since this specification was adopted while the finalization process of MOF 2.0 Core was not completed, there might occur inconsistencies with respect to these documents. In these cases, the following standards serve as normative reference:

- Object Management Group, MOF 2.0 Core Final Adopted Specification, ptc/03-10-04.

- Object Management Group, UML 2.0 Infrastructure Final Adopted Specification, ptc/03-09-15.

4 Additional Information

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

- Fraunhofer Institute FOKUS
- IKV++ Technologies AG
- Technical University Berlin

With support from participants of the MASTER project consortium supported by the IST Program of the European Union:

- European Software Institute
- THALES ATM
- THALES Research and Technology

The technology proposed by this specification is based on the work of the MASTER project (http://www.esi.es/Master) of the IST Program of the European Commission. The authors would like to thank the participants of this project for their contributions and review activities.

5 Methodology

This chapter describes the authors' approach for the MOF 2.0 IDL language mapping and assumptions about the MOF model. Due to the fact that the mapping is strongly dependent on the MOF 2.0 Core and other currently ongoing RFP processes (such as MOF2.0 QVT RFP [4] MOF2.0 Versioning and Development Lifecycle RFP [5]…), this specification follows its own approach for the definition of the language mapping and outlines the vision, how this language mapping will fit in emerging standards and OMG's MDA (for a detailed specification refer to Chapters 6 and 7).

5.1 Overview

The key to the authors' approach was to divide the mapping into two parts, along with the orthogonality of the concepts in the MOF metamodel framework:

- A MOF Model core mapping, allowing usual constructs to map onto IDL. Thereby, the mapping provides further two compliance points (i.e., EMOF and CMOF compliance).
- Several standardized MOF Services (Utilities) with specific mappings, allowing for customized IDL (and computational semantics). These separate mappings are orthogonal extensions to the core mapping and provide, for example, model management facilities for the generated repositories.

5.2 Assumptions of MOF 2.0 Core

Based on the latest revised adopted specification to the MOF 2.0 Core document and the discussions with the MOF2 FTF, our assumptions of the MOF 2.0 Core are as follows:

- MOF 2.0 will reuse the Core packages from the UML 2.0 InfrastructureLibrary, wherefore the mapping is (at least) aligned to those concepts that are imported by MOF. Furthermore, we assume MOF to comprise the following packages (see Figure 5.1):
    - InfrastructureLibrary::Core::Basic
    - InfrastructureLibrary::Core::Constructs
    - InfrastructureLibrary::Core::Abstractions
    - InfrastructureLibrary::Core::PrimitiveTypes
    - MOF::EMOF (as result after package <<*combine*>>)
    - MOF::CMOF (as result after package <<*merge*>>)
- Based on the current revised adopted specification to the document for the MOF 2.0 Core RFP (see [8]), we provide a mapping for the modeling elements defined therein (details listed in Section 5.3.2, "MOF Model," on page 5.).
- Integer and String primitive types are constraint in MOF (see section 9.4 of [8])
- Visibility is ignored completely in MOF models.
- Enumerations do not have attributes or operations.

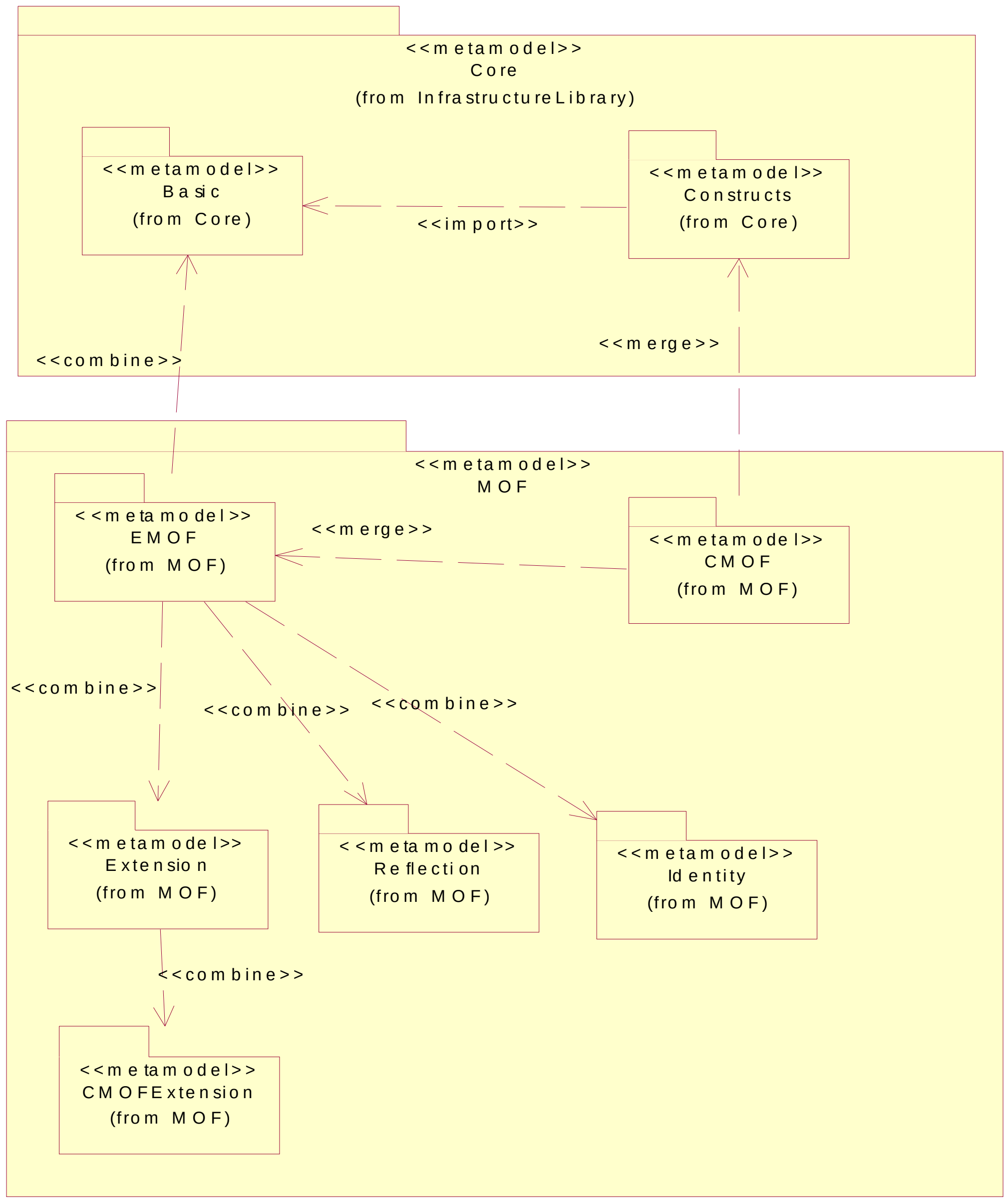


Figure 5.1 - MOF Model Used for the Definition of the Mapping

5.3 Mapping MOF to the IDL Metamodel

In terms of MDA, the MOF to IDL mapping is a model transformation from the MOF metamodel to the IDL metamodel. Thus, this section introduces the concepts that are needed to transform (meta-) models and it outlines the approach of the separation of the core mapping from the MOF Services.

5.3.1 Separation of MOF and Service Mapping

As mentioned earlier, the approach divides the MOF 2.0 IDL mapping into two separate mappings: one for the core package and several others for what are called the MOF Services; supposing that the MOF package consists of the MOF Model as is (a subpackage called MOF Model contained by the MOF package) and less or more other packages, which are logically orthogonal concepts (utilities), providing services for the MOF Model and which are itself modeled by concepts from the core.

Specifying multiple roles or views of elements allows specialized behavior of interfaces. This allows modelers to freely configure their mapping by importing special semantics while remaining standard conformant through well defined rules in an extension framework.

The solution to these arising problems is to use CCM component-facets, where each offered service is mapped onto a facet that is supported by the component (see Section 6.1.2, “The CCM Mapping,” on page 11).

Thus, the computational semantic of metamodel instances within a repository maintain a common core - which is defined using abstract interfaces supported by a component – and, further on, are open to provide additional facilities as kind of service.

5.3.2 MOF Model

The core-mapping is a revised version of the current IDL mapping introducing abstract interfaces and valuetypes. It is based on the elements imported from UML 2 InfrastructureLibrary (from UML adopted document [9]) Core packages combined into the EMOF and CMOF packages (cp. [8]).

The EMOF and CMOF packages are constructed by the deep-copy merge algorithm described in [8], section 9 (*PackageMerge::define*). The resulting modeling elements are listed below. A proposed mapping onto IDL elements is provided later in this specification in Chapters 6 and 7.

EMOF:

- Class
- Enumeration
- EnumerationLiteral
- Extent
- Operation
- Parameter
- Package
- PrimitiveType
- Property
- Tag
- URIExtent

Additional elements of CMOF (non-abstract):

- Association
- Comment
- Constraint
- DataType
- ElementImport
- Exception
- Expression
- OpaqueExpression
- PackageImport
- PackageMerge

5.3.3 Reflection

Reflection is regarded as kind of service and maps differently from the main mapping (since MOF2 contains two Reflection packages). In this specification, the Reflection service is used to exemplify the adaptiveness and flexibility of our approach.

The Reflective operations in [8] (section 10) served as a fundament for the API contained in this specification while at the same time aligning these definitions with build-in features of our preferred target: CORBA components.

More on reflective support can be found in Section 6.3.6, “Support for the Reflection Service,” on page 30 and details in Chapter 8.

5.3.4 Notation and Transformation

After reviewing the submissions to the MOF2.0 QVT RFP [4], the submission proposals can be roughly classified into rule-based and declarative approaches to the transformation issue. While the former describes *how* a target model is created (using rules, some kind of create operations and other, imperative constructs), the second simply states, *what* has to exist at the end of a transformation (some kind of post-condition combined with a creation-semantics for the evaluation of the declarations).

We believe that both approaches are semantically (and at least computationally) equivalent, but differ in their way of being expressive, intuitive, or how they allow for implementation derivation (not to mention aspects of traceability, reuseability, or user-interaction).

Therefore, our proceeding is to reuse the Object Constraint Language (OCL) for the purpose of defining transformations as invariants between metamodels. This approach is in the same conceptual line as e.g., the QVT submissions [11] and [12], but will not anticipate or influence the standardization process.

5.3.5 Combining models using invariants

The idea behind a combination using invariants is based on the hitherto modeling principle of classes and associations (e.g., MML [4], [11] or [12]). Source model (here: MOF) and target model (here: IDL MM) are stuck together in one model, using associations starting at elements of the first model and ending at elements at the second. These associations are the abstract view of the mapping.

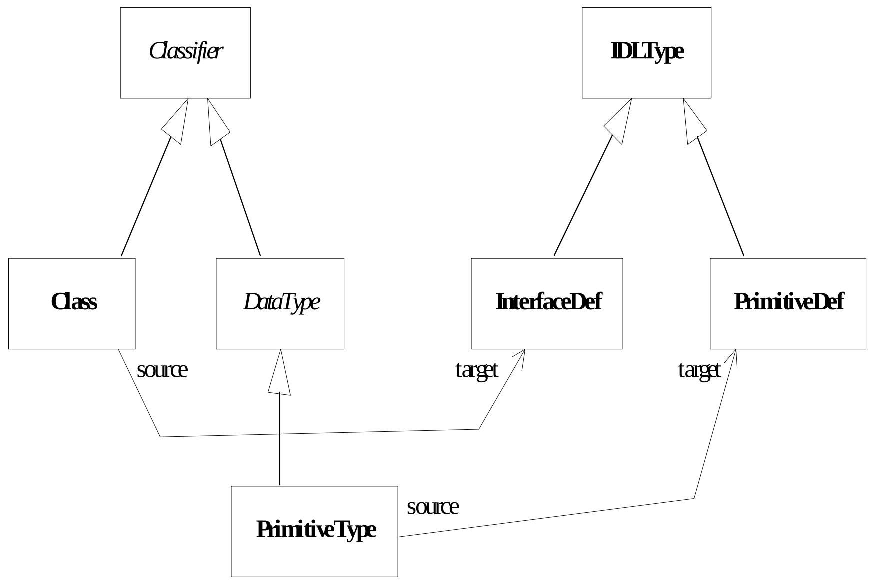
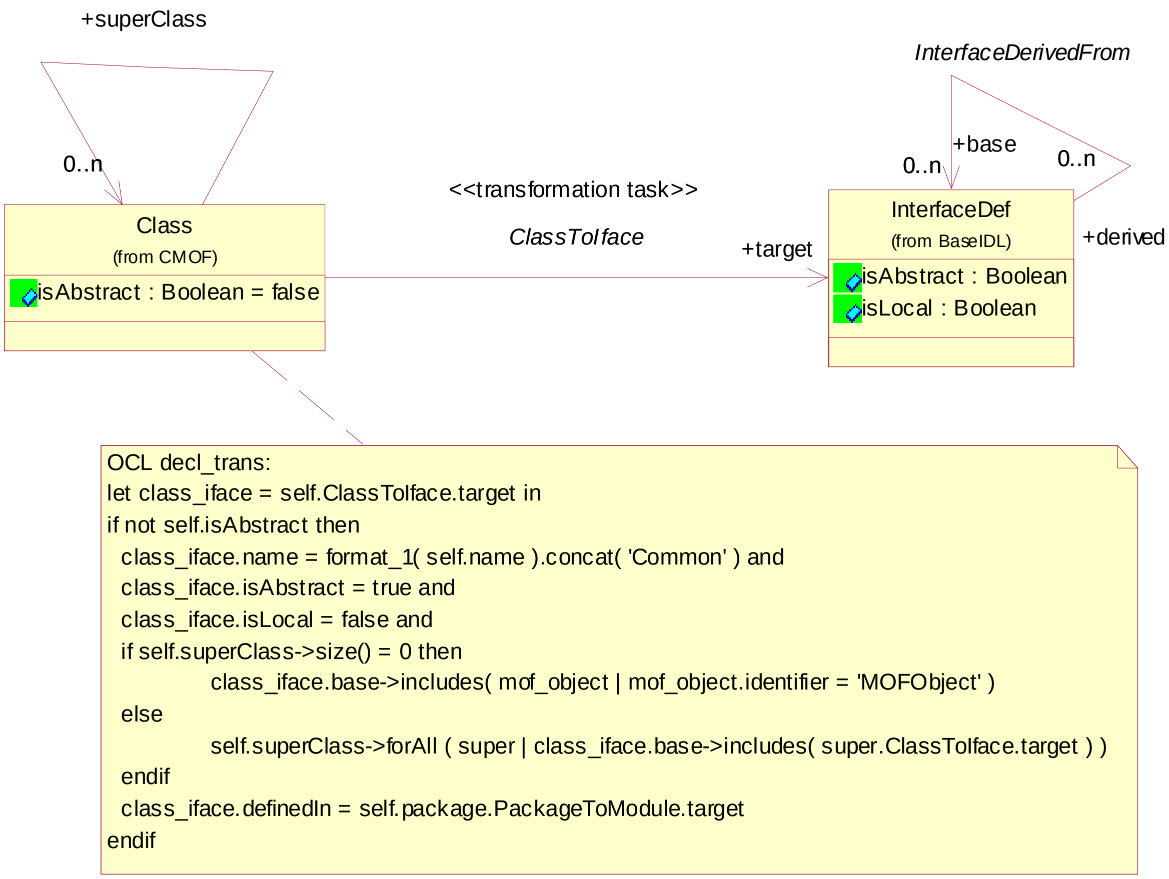


Figure 5.2 - Class-to-Class mapping using associations

Note that the associations in the diagram are directed with multiplicity one-to-one. The semantic of this kind of relationship carries out the fact, that for each instance of a class of the source model, in most cases one instance of the target model is generated. The multiplicity can be referred to as an instance-level expression in the terminology illustrated above. In most cases, attention must be paid if elements are related, because of e.g., containment-consistency (containment relationships of elements in the source model may lead to similar relationships in the target model).

The instance-level definitions are straightforward using OCL constraints expressing the language mapping as invariants of the relationship.

Moreover, a specialized notation with a kind of <<instanceOf>> stereotype will ease the notation and specification of the mapping, factoring out **forAll** operators and further logic operations. In most cases, OCL constraints in the (OCL-) context of the source element will suffice (using the OCL ***self*** keyword), navigating from source instances to target instances via the association ends (see diagram for a more complex example).



```
OCL decl_trans:
let class_iface = self.ClassToIface.target in
if not self.isAbstract then
  class_iface.name = format_1( self.name ).concat( 'Common' ) and
  class_iface.isAbstract = true and
  class_iface.isLocal = false and
  if self.superClass->size() = 0 then
            class_iface.base->includes( mof_object | mof_object.identifier = 'MOFObject' )
  else
            self.superClass->forAll ( super | class_iface.base->includes( super.ClassToIface.target ) )
  endif
  class_iface.definedIn = self.package.PackageToModule.target
endif
```

Figure 5.3 - Class-to-interface enhanced with Instance-to-Instance diagram

The main advantage of using associations for the mapping, is to reuse this common concept without the need for any semantic extensions; but immediate drawbacks are: (1) it is hard finding the complete OCL constraints and models will become complex, (2) this notation is not much constructive and one has to re-interpret the association's semantic for code generation of mapping-generators, and (3) concepts of parameterization, pattern definitions, or traceability concerns are not coped.

NOTE: Since a specification for defining transformations between models is still not adopted, the definition of the MOF2.0 to IDL/CCM mappings as model transformation must be adjusted to the final transformation standard. The OCL constraints for MOF2IDL are provided in a non-normative document [13].

5.4 MOF Services

What we call service is not a service in the general sense. It is more a utility or pattern. MOF Services are not a totally new concept, nor the reinvention of CORBA Services. The truth is in the middle: what we call MOF Service may be an abstraction of common object services raised to the model-level to provide the reuse of current methods in model engineering, while combining them with pattern oriented approaches such as design-patterns. They may be found either by reverse-engineering of existing object-services (low-level services) or by specifying kinds of general methods (high-level services). Some typical representatives are:

- Reflection (see Section 5.3.3, “Reflection,” on page 6)
- Versioning and User Management
- Transactions
- LifeCycle
- ActiveRepository (Event based communication, see Section 6.3.7, “Support for Notification based Model Change Communication,” on page 40)
- Additional TypeLibraries (e.g., FloatingPoint types)
- OO-Design Patterns
- …

As denoted, our vision of these services goes beyond basic capabilities, such as a TypeLibrary or a Reflection service: services could enhance current (meta-) modeling as a kind of library of concepts, ready for reuse and integration in both, models and metamodels. Services bridge between the generation of concepts out of a model and the reuse of existing components on a platform as kind of library.

Problem-specific mappings as requested by many vendors challenge the standardized all-in-one MOF mapping solution so that a customized approach would be preferable: remaining standard conformant while simultaneously extending to your own needs. In most cases, metamodel repositories essentially need some basic facilities in form of ready-to-use interfaces like transactions, versioning, or user-management.

Furthermore, currently, every change to the MOF-to-IDL specification results in a change in the mapping itself. Vendors may propose changes (as already done) to the mapping in a future release of MOF, but seriously, the core of the problem is, that MOF is - until now - not *open enough* in the sense of allowing (independent) extensions to take place.

If there exists a possibility to specify a kind of service afterwards (with an appropriate integration strategy in the technology framework), then this problem could be solved smoothly (without release- or mapping changes). Furthermore, if this service is specified once, the domain-specific mappings may port the concept to other platforms apart from CORBA IDL (i.e., integration with or transformation to the .NET metamodel or other (component) models).

However, what we had in mind when designing the mapping was what we call an *ActiveRepository Service*, which allows repository initiated communication by notification of events (details in Chapter 6 - Specification). Applying such a drastic change to MOF as a service outside the core will prove the extension framework and verify our approach of separation of concerns.

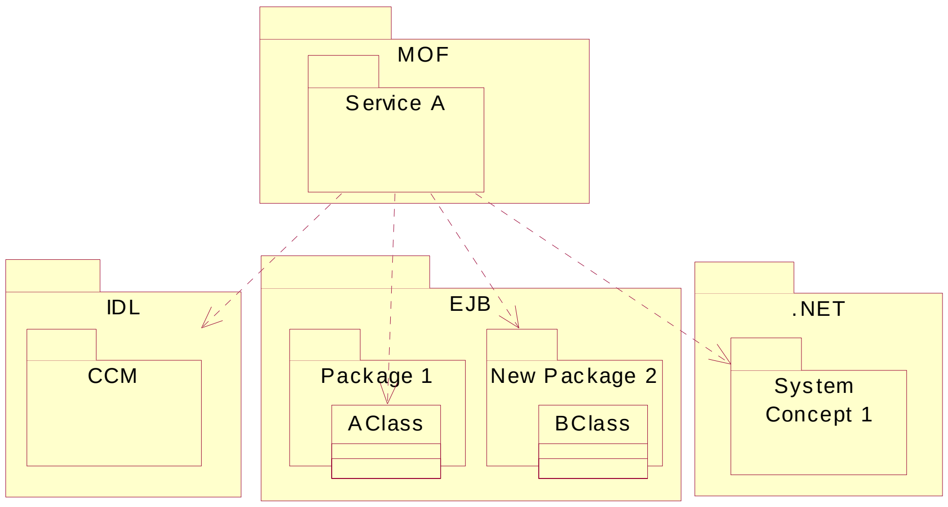


Figure 5.4 - A MOF Service maps onto specific domains

6 Specification

6.1 Overview

This document defines a complete set of mapping rules for all elements of the MOF 2.0 model. This chapter outlines the basic principles of the MOF 2.0 IDL mapping approach, focusing on the advantages the approach has. The approach targets at an improvement of the current mapping while at the same time using the concepts of the CORBA Component Model as one possible realization.

Because MOF2 is adopted, but not yet finalized, this specification refers to the MOF2.0 Core submission document (ad/03-04-07) [8] and to issues discussed in the FTF. Since MOF2 reuses the main parts of the Core package of the UML2 InfrastructureLibrary, outstanding changes will be rather small. Because the Core specification defines basically two compliance points (i.e., EMOF and CMOF) the IDL mapping also supports these compliance points.

Nevertheless, the general idea of the mapping is a two-fold specification of one mapping profile for the CORBA Component Model as target and one profile for the normal; (Base-) IDL. The realization is done by a splitting of the mapping into three parts.

- A Common Mapping, which provides the main derivation of meta-model elements to IDL. This part of the mapping is a common subset (intersection) of the following mappings.
- A *CCM Mapping* completing the Common Mapping for the CORBA Component architecture. Additional components are derived for the concrete access of the interfaces derived in the Common Mapping.
- A *Base-IDL Mapping* that provides the derivation of non-component aware IDL interfaces. It is in the same conceptual line as the CCM Mapping, but focuses on native CORBA objects in the style of the present MOF1.4 mapping.

6.1.1 The Common Mapping

The Common Mapping is the core of the IDL mapping. On its own, this mapping is incomplete and serves only as a common base for further mappings. It targets a structural and behavioral specification of model elements. An outline of the derived IDL is as follows:

- MOF model elements that provide structural or behavioral features are mapped to abstract interface declarations. These interfaces correspond to the presently derived instance interfaces, but also support call-by-value and call-by-reference semantics.
- Furthermore, IDL valuetype definitions are derived, that are declared to support these abstract interface definitions - instances of these valuetypes can be used to set/update the attributes of a class instance or to notify changes of attribute values to clients.

6.1.2 The CCM Mapping

Based on the interfaces and valuetypes derived in the Common Mapping, the CCM Mapping completes these IDL definitions with full component support. In detail:

- MOF model elements of type class, package, and association are mapped to CORBA component definitions.
- Structural and behavioral features of these model elements, that are mapped to abstract interface declarations in the Common mapping, are declared to be supported by the respective CORBA components.

- There will be no class proxy or package factory interfaces, the semantics of these interfaces (as they are defined in the MOF 1.4 IDL mapping) is implemented by home interfaces of the respective components.
- Besides reflection, an additional service is proposed: the notification based communication of (sub)model changes initiated by the repository. In general, services are mapped to behavioral features of the respective component (reflection maps to a facet, notification based model change maps to event ports).

Examples regard the possibility to order receptacles of type <multiple use> and issues with primary keys of entity type components.

6.1.3 The Base-IDL Mapping

Besides the CCM Mapping, a non-component mapping is specified to deal with component-unaware clients and to handle downward-compatibility issues. Since the equivalent IDL of components does not suffice by means of interface structure and operations, this extra mapping is essentially needed. In more detail:

- Additionally to the abstract interfaces and valuetypes derived for elements of a meta-model in the Common Mapping, non-abstract interfaces are added, inheriting from the abstract interfaces and extending these with functionality needed in a non-component environment.
- For instance, management and access - the former Class Proxy interfaces are waived and functionality of these interfaces is combined into the package interface.
- An additional "Query" interface is derived for the querying of models. This interface may provide operations for issues to be specified in the Query/View/Transformation (QVT) RFP.

6.2 Common Mapping Rules

The common mapping rules are the intersection of the CCM- and Base-IDL mapping. The rules concentrate on the structural and behavioral specification of model elements in IDL, leaving the concrete access via object- or component-references to the CCM and Base-IDL Mappings.

6.2.1 Mapping of Identifiers

Identifiers of elements of a MOF 2.0 compliant model are mapped to CORBA IDL identifiers, in this specification we import the rules to generate IDL identifiers from section 5.7.1 of MOF 1.4 [1]. We define the following pseudo operations that are used in the mapping specifications:

- **identifier format_1 (<MOF model element identifier>)** returns an identifier for an IDL definition according to 5.7.1.2 of [1],
- **identifier format_2 (<MOF model element identifier>)** returns an identifier for an IDL definition according to 5.7.1.3 of [1],
- **identifier format_3 (<MOF model element identifier>)** returns an identifier for an IDL definition according to 5.7.1.4 of [1],
- **identifier concatenate (string, string)** returns a concatenation of the two parameters.

For the resolution of a name in the target IDL, basically the naming conventions of MOF, v1.4 are reused (interfaces, that are derived for a class, receive an identifier in format 1, operations in format 2 and so on). For details see the following sections.

6.2.2 Globals

To provide a common base interface for every derived MOF model element, we introduce an (abstract) base interface for having fundamental operations such as **delete** and **is_equal** available.

Furthermore, this new base interface can compensate the dropping of the default reflective interfaces. Thus, the new common base-type is (omitting exceptions):

```
abstract interface MOFObject {
    MofId get_mof_id();
    boolean is_equal( in MOFObject other_object );
    void delete ();
    //...
    void set_value_depth( in unsigned long depth );
    unsigned long get_value_depth();

    void set_call_semantic ( in boolean as_value );
    boolean get_call_semantic ();
};
```

The **is_equal** and **delete** operations are needed for identity support and destruction of objects, since abstract interfaces do not inherit from **CORBA::Object**[1]. For more details refer to Section 6.2.13, “Mapping of Reflection,” on page 25 and Chapter 7.

Additionally, the **MOFObject** interface provides every object with operations to determine the composition/assembly when querying an instances state. Thus, clients can adjust what they want to receive from the repository: references or valuetypes. These operations were meant to meet the requirement of minimizing the amount of remote calls to a repository. More details can be found in Chapter 7.

Since the following sections will introduce CORBA valuetypes, we provide also a common base-type for these valuetypes:

```
valuetype MOFState supports MOFObject {
    private MOFObject origin;
};
```

As shown later, the base-valuetype is essentially needed with respect to state update of model instances.

6.2.3 Mapping of the PrimitiveTypes Package

Intuitively, the **InfrastructureLibrary::Core::PrimitiveTypes** are mapped to the corresponding CORBA IDL types. This is aligned with the constraints of CMOF as described in section 9.4 of [8] (CMOF Constraints) for the primitive types.

1. So there is no possibility to override existing operations, for example, **is_equivalent** (or other ways to introduce MOF computational semantics.

InfrastructureLibrary::Core::PrimitiveTypes	**CORBA IDL types**
Integer	long
String	wstring
Boolean	boolean
UnlimitedNatural	unsigned long

Besides these four basic types every instance of the class PrimitiveType (so called user-defined primitive types) in a MOF compliant model is not mapped to IDL.

Rule (1) Apart from the types defined in the PrimitiveTypes package, instances of the class PrimitveType are not mapped to a specific IDL construct. Instead, whenever a model element refers to a user-defined primitive type in a metamodel, the full qualified name of this instance is generated.

Afterwards, it is up to the modeler to provide, for example, an appropriate IDL **typedef** definition in the module to define the primitive type.

6.2.4 Mapping of Collections (MultiplicityElement)

In MOF, v1.4, collections are mapped to IDL sequences. If, for example, an attribute of a class in the model defines an attribute of type **CorbaIdlTypes::CorbaUnsignedLong**, with multiplicity [1..n], non-ordered and unique, the resulting IDL operations to set and get the attribute values are defined as:

```
module CorbaIdlTypes {
    typedef sequence < unsigned long > ULongSet;
};
// …
void set_attribute ( in ULongSet value );
ULongSet get_attribute ( );
// …
```

The problem with this is that usually large collections need to be communicated between a repository and its clients. Worse than that, it is the only way for a client to access this value.

To support a more convenient access to collections, we generate specific CORBA IDL iterator definitions. We use the concept of abstract interfaces to let the client choose whether to access the collection by value or by reference.

We do not distinguish the operations for bag/set/list/ulist in this section. This will be done in the specification details in Chapter 7.

Rule (2) For all elements declared in a model, IDL collection types are generated. The rules for the production follow, using a template notation, where <type of collection> is the IDL type generated for the model element, and <collection kind> is one of the kind suffixes "**Bag**," "**Set**," "**List**," "**UList**," following the rules of 5.7.2 of MOF, v1.4 [1].

typedef sequence < *<type of collection>* **>** *<type of collection><collection kind>*;

Rule (3) An iterator abstract interface is generated to access the values of a collection. The interface declares the operations:

- **get_value** to access the particular value the iterator points to in the collection.
- **insert_value** and **modify_value** to set the particular value the iterator points to in the collection.
- **remove_value** to remove the particular value the iterator points to.
- **next_one** to get an iterator pointing to the next value in the collection.
- **previous_one** to get an iterator pointing to the previous value in the collection.
- **begin** to get an iterator pointing to the first value in the collection, and
- **end** to get an iterator pointing to the last value in the collection.
- **is_empty** to test whether the collection is empty.

```
abstract interface <type of collection><collection kind>Iterator:
MOF::IteratorBase {
    <type of collection> get_value () raises (MofError);
    void insert_value ( in <type of collection> value ) raises (MofError);
    void modify_value ( in <type of collection> value ) raises (MofError);

    <type of collection><collection kind>Iterator next_one()
                    raises (MofError);
    <type of collection><collection kind>Iterator previous_one()
                    raises (MofError);

    <type of collection><collection kind>Iterator begin()
                    raises (MofError);
    <type of collection><collection kind>Iterator end()
                    raises (MofError);
    boolean is_empty();
};
```

Rule (4) A CORBA IDL valuetype definition named <type of collection><collection kind>**AsValue** is being produced supporting the abstract interface generated following Rule (3). This valuetype contains a private member of type <type of collection><collection kind> named **value**. The valuetype is being used to receive the collection as value, containing all elements of the collection. Besides the private state member, it gets a factory operation named create, with an in-parameter of type <type of collection><collection kind>.

```
valuetype <type of collection><collection kind>AsValue
    supports <type of collection><collection kind>Iterator {

    private <type of collection><collection kind> value;
    factory create ( in <type of collection><collection kind> initializer );
};
```

Rule (5) Furthermore, a "concrete" iterator interface is being generated, inheriting from the abstract interface produced as defined by Rule (3). This interface declares one operation named **as_value**, that returns a valuetype generated following Rule (4).

```
interface <type of collection><collection kind>AsReference
        : <type of collection><collection kind>Iterator
{

    <type of collection><collection kind>AsValue as_value();
};
```

Later, all operations to get access to attributes with IDL type of collection, will use the abstract interface declaration as return type. With this construct the client can decide whether he wants to access a collection by value or by reference.

The inheritance of the abstract base interface **IteratorBase** is needed to specify an iterator as read-only:

```
abstract interface IteratorBase {
    readonly attribute boolean read_only;
};
```

Moreover, it is intended to reuse this collection pattern every time an instance of class MultiplicityElement is derived, so that clients simply can get an iterator instance (by reference or by value) and iterate over the collection.

An example is given in Section 6.2.17, “Mapping of Attribute and Operation Constructs,” on page 27. For specification details, see Section 7.4, “Exceptions,” on page 49.

6.2.5 Mapping of Construct Package to IDL Modules

Intuitively, packages of a MOF model will be mapped to IDL modules. Nested packages are mapped to IDL modules defined within the module of the container package.

Rule (6) A Package of a MOF 2.0 compliant model is being mapped to a CORBA IDL module definition with the identifier **format_1 (<package identifier>)**.

Rule (7) If the package is nested (i.e., contained in another package), the CORBA IDL module is defined in the scope of that module that was generated for the container package.

For specification details, see Section 7.5.1, “EMOF::Package,” on page 50.

6.2.6 Mapping of Construct Class

Classes of a MOF model are mapped to abstract interfaces declaring IDL operations for the structural and behavioral features. Additionally, for the representation of a class's state, an IDL valuetype is derived.

Rule (8) A Class of a MOF 2.0 compliant model is mapped to a CORBA IDL abstract interface, named **concatenate (format_1 (<class identifier>), "Common")**, in the scope of the IDL Module, that is generated for the package construct the class is defined within. This interface is further referred to as the common interface and inherits from a common base interface named **MOFObject** (only if its class has no super class).

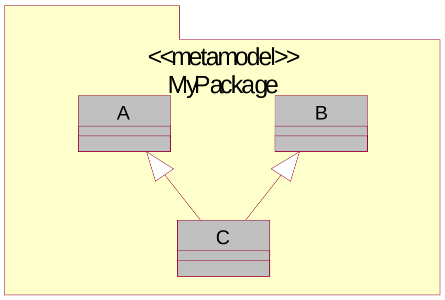
Rule (9) In the same module, a CORBA IDL valuetype is declared with the name **concatenate (format_1 (<class identifier>), "State")**, that supports the abstract interface generated following Rule (8) and inherits from **MOFState (truncatable)**.

6.2.7 Mapping of Inheritance between Classes

Inheritance of classes of a MOF model is mapped to inheritance of the abstract interfaces generated for these classes.

Rule (10) If classes in a model inherit from another, this relation is mapped to inheritance relations between the respective abstract interfaces generated according to Rule (8). The valuetypes generated according to Rule (9) do not inherit from each other.

Example (1) Let **A**, **B** and **C** be classes in the package **MyPackage**. **C** inherits from **A** and **B**. In this case, the following IDL definitions are generated:



```
module MyPackage {
    abstract interface ACommon : MOFObject { };
    abstract interface BCommon : MOFObject { };
    abstract interface CCommon
      : ACommon, BCommon { };
    valuetype AState : truncatable MOFState
      supports ACommon { };
    valuetype BState : truncatable MOFState
      supports BCommon { };
    valuetype CState : truncatable MOFState
       supports CCommon { };
} ;
```

6.2.8 Mapping of Attributes and Operation Constructs

The approach here is to define IDL operations for attributes (instances of the MOF model class Property) and operations of a class within a MOF model. For attributes of a MOF model class, these IDL operations can be used to set, retrieve, and modify the value of that attribute. The derivation rules are enhanced later on for the CMOF construct of Redefinition (see Section 6.2.12, "Mapping for Redefinitions," on page 23).

We refer to the type of an operation or attribute as the type that is derived for that element (common interface, primitive type, or data types). Moreover, we do not distinguish navigable association ends and attributes of type class (cp. to UML2 Infrastructure and see discussion in Section 6.2.11, “Mapping of Associations,” on page 22, and Navigability and owned Properties).

Rule (11) The generated IDL operations for attributes/operations of a MOF model are defined within the scope of the abstract interface (common interface) generated for the containing class. Furthermore, the type chosen for parameters, etc. is the type that is derived for its corresponding model element type.

Rule (12) For a non-derived attribute with multiplicity [1..1], two operations to set and get the attributes value are declared - taking into account Rule (11). In case of attributes that have the isDerived property set to true only the get operation is generated. For optional attributes with a multiplicity of [0..1], an additional operation to unset the attribute is declared.

The operation to get the value of the attribute has the name **concatenate ("get_", format_2 (<attribute identifier>))**. The return type is the IDL type generated for the attribute’s type in the model. The operation to set the value has the name **concatenate ("set_", format_2 (<attribute identifier>))**. The operation has one parameter of the IDL type generated for the attribute’s type in the model. For optional attributes the unset operation receives the name **concatenate ("unset_", format_2 (<attribute identifier>))**.

One reason for generating an IDL valuetype according to Rule (9) is to be able to communicate the whole state of an object in the generated repository with one (or only a few) operation calls. Therefore, the concrete value of an attribute needs to be a state member of this generated valuetype.

Rule (13) Furthermore, the IDL valuetype generated following Rule (9) is extended by a private state member of the IDL type generated for the attribute’s type with the name **format_2 (<attribute identifier>)**.

Rule (14) The definition of Rule (13) will be recursively applied to the attributes of all super classes of a class in the model. This results in an IDL valuetype definition containing state members for all non-derived attributes of the respective class and their super classes.

The state members of the valuetype are always private. The reason is that the operations to access the state members are already defined by the abstract interface the valuetype supports, and the actual data should be hidden.

An important aspect with respect to the state of an instance is how associated elements - referred to via properties of classifier type - are assembled into the valuetype. Our solution is that this can be adjusted dynamically for each instance through the inherited **MOFObject::set_value_depth** operation. Thereby, the valuetype will contain valuetypes recursively up to a specified number **n** (passed as parameter) viewing associations as kind of “hop-count.”

Furthermore, the attribute mapping has to consider collections. Since the iterator mapping (see Section 6.2.4, "Mapping of Collections (MultiplicityElement)," on page 14) also applies for attributes with multiplicity upper > 1, we extend the mapping rule defined by Rule (12) with the following:

Rule (15) For an attribute with multiplicity upper>1, two operations to set and get the attribute value are declared - taking into account Rule (2), , and Rule (11).

The operation to get the value of the attribute has the name **concatenate ("get_", format_2 (<attribute identifier>))**. The return type is of the IDL collection type generated for the attribute's type in the model according to Rule (3) (abstract iterator interface). The **MOFObject::set_call_semantic** operation is used by the client to decide, whether the operation shall return the collection by value (if **as_value** is true) or as reference (if **as_value** is false).

Rule (16) The operation to set the value has the name **concatenate ("set_", format_2 (<attribute identifier>))**. The operation has one **in**-parameter of the IDL collection type generated for the attribute's type in the model according to Rule (3).

Furthermore, we extend the definition of Rule (13) for the case, that the multiplicity has an upper bound > 1, the IDL valuetype generated according to Rule (9) is extended by a private state member with the name **format_2 (<attribute identifier>)**. The type of the member is the IDL collection type generated for the attribute type in the model following Rule (4).

For specification details, see Chapter 7. A comprehensive example especially for the multiplicity is given in Example (5).

Property modifications

This subsection gives some rules of thumb about what is altered in the normal mapping (i.e., (Rule (11) to Rule (15)) with respect to attributes and associated elements of a property in MOF2. The specification details can be found in Chapter 7.

Rule (17) If an instance of class Property has a property set to one of the following, the mapping is altered as described:

- **{isReadOnly = true}** : for a read-only property, no set-operation is derived. Instead, the value must be specified when creating the instance that contains the property (create operation with parameters, see Rule (34)/Rule (51)). However, the attributes state may be changed by operations that exist for the instance.
- **{isDerivedUnion = true}** : if a property is a derived union, then the "get" operation is renamed to **concatenate ("union_", format_2(<prop_name>))**. Depending on the multiplicity kind, the return type is either the type derived for the property's type, or in case of a collection typed to the (read-only) iterator that is derived for its multiplicity type (cp. Rule (3)). The computational semantic of the operation is equivalent to the UML2 abstract semantic of unions (i.e., the returned set of instances depends on the **subsets** defined in subclasses).
- **{subsettedProperty->size() > 0}** : defining a property as subset leads to the derivation of the usual **get** and **set** operations, but with the identifiers **concatenate ("get_subset_", format_2 (<prop_name>)**) and **concatenate ("set_subset_", format_2 (<prop_name>))**, respectively. The computational semantic of these operations is

equivalent to the UML2 abstract semantics, the parameters and return type according to Rule (12) and Rule (15). Note that this implies that the (sub-) set of instances for this property is included in the returned set of instances of an existing **union**-operation of superclasses for which this property is a subset.

- **{redefinedProperty->size() > 0}** : redefinition of an element is covered in Section 6.2.12, "Mapping for Redefinitions," on page 23.
- **{isID = true}** : if a class owns a property with the isID flag set to true, the computational semantic of the inherited **is_equal** operation is altered (from **MOFObject**). In this case, identity is established via this property (the MOF2 abstract semantic should be refined and reused for the mapping). We interpret this flag as follows:

  1. If the type of the property is a primitive or data type, two instances are identical if the values are the same.

  2. If the property has a classifier type, two instances are identical if they reference the same MOFObject.

  3. If the property has a constructed (data-) type or a collection type, two instances are identical if and only if all of the above holds true recursively for all elements in the composition/ collection of the property. Note that this may result in checking a whole subtree of elements for their identity!
- **{isComposite = true}** : The composite flag has influence on the computational semantic of the delete operation of this property, since it establishes a part-of relationship. If its type is of classifier type, then the delete operation also destroys these referenced instance(s). We intend to reuse the MOF1.4 delete semantic for composition.
- **default value** : the string-typed attribute **default** is underspecified in the current MOF2.0 Core document, because it cannot be estimated that the string parameter is parsed for a valid initial value (it is only useful for visualization). Therefore, we do not map its value (see note).
- ***{opposite->size() = 1}*** : if an attribute has an opposite, these two properties are entangled and constrained with respect to their computational semantics. They behave exactly the same as a bidirectional navigable association and therefore represent the mutual knowledge for the instances of each other.

6.2.9 Mapping of DataTypes and Enumerations

The mapping for data type instances including subclasses is straightforward and in the same line as the class mapping.

Rule (18) Instances of class DataType are mapped to valuetype definitions and abstract interface within the module that is derived for its containing package according to Rule (6). The valuetype gets the DataType's name as identifier in **format_1**. The valuetype has no supertypes and supports the abstract interface (which identifier is **concatenate (format_1 (<data_type name>), "Operations")**). Inheritance of data-types is realized by inheritance of the abstract interfaces (see Section 6.2.7, "Mapping of Inheritance between Classes," on page 17).

Rule (19) Furthermore, the valuetype contains a **private** member for each contained ownedAttribute of the DataType and all supertypes recursively with identifier **format_1 (<owned_attribute_name>)** and type of the attributes derived IDL type.

Rule (20) If the DataType contains operations or properties, these are mapped into the abstract interface with identifier **concatenate (format_1 (<data_type name>), "Operations")** analogous to the Rule (11) and Rule (12). Furthermore, the valuetype derived in Rule (18) always contains two operations:

- **boolean is_equal (in <DataType> other)** : This operation allows for the comparison of two instances of this data type by value (deep compare of members).
- **factory create (in <MemberType> <member>, ...)** : The parameter list contains an in-parameter for each of the ownedAttributes of the DataType instance (in the same order) of the type, that is derived for that attribute.

Rule (21) Since enumerations are constrained to have no operations or properties (cp. constraints in [8]), instances of class Enumeration are mapped to an IDL **enum** (defined in the module that is derived for the containing package according to Rule (6)) with identifier **format_1 (<enumeration_name>)**. Each contained EnumerationLiteral instance of the enumeration becomes an identifier of the enum (**format_1 (<enumeration_literal_name>)**); the order is the same as in the *ownedMember* collection).

Rule (22) In case of inheritance of enumerations, the inheriting enumeration obtains all literals as identifiers in **format_1 (<enumeration_literal_name>)** of all base-enumerations recursively.

Since inheritance of enumerations maps to type-unrelated **enum** definitions in IDL, clients must beware of using the "right" scope when, for example, calling an operation that requests an enum as parameter (see also Section 6.2.12, "Mapping for Redefinitions," on page 23).

6.2.10 Mapping of Exceptions

The lack of an IDL exception not being able to inherit from other exceptions[2] leads to the decision, to reuse the exception framework of MOF1.4. With some minor improvements, the elementary exception remains still **MofError**:

2. And moreover the inability of the **raises** clause of operations to throw other data types than exceptions.

```
struct NamedValueType {
    wstring name;
    any value;
};

typedef sequence < NamedValueType > NamedValueList;

exception MofError {
    string error_kind;
    MOFObject element_in_error;
    NamedValueList extra_info;
    wstring error_description;
};
```

Since the raisedException property of the Operation class can reference everything that is of type Class, the exception mapping is more complicated than in MOF1.4.

Rule (23)	If an operation definition raises an exception of type Class, the operation's **raises** clause obtains a **MofError** statement. In the exceptional case, a MofError exception is raised with the **MOFObject element_in_error** member set to the appropriate instance.

6.2.11 Mapping of Associations

An association defined in a CMOF compliant model may or may not be mapped to CORBA IDL definitions (the configurability is a matter of parameterization within the transformation language and thus out of the scope of this specification). Since properties of classes with class-type are semantically equivalent to navigable association ends, the definitions of Section 6.2.8, "Mapping of Attributes and Operation Constructs," on page 17 are reused for navigable association ends.

In addition, if an association is mapped to IDL, then the following is produced:

Rule (24)	If an association construct of a model is mapped to CORBA IDL, an abstract interface with the name **format_1 (<association name>)** is generated. This abstract interface defines operations similar to those specified in section 5.8.10 of MOF1.4 and also inherits from **MOFObject**. The derived operations are described in Section 7.6.3, "CMOF::Association," on page 59.

Rule (25) Furthermore, an IDL struct for the link-set is derived for an association with name **concatenate (format_1 (<association name>), "Link")**. This struct contains two members of type of the association's ends' derived abstract (common) interfaces with the same name as the association end (in format_2). The type of the members is set to the abstract interfaces derived according to Rule(8). Furthermore, a **typedef sequence** of type **<association-name>Link** with name **<association-name>LinkSet** for the link-set is also generated.

Rule (26) Additionally, a CORBA valuetype is derived with identifier **concatenate (format_1 (<association name>), "State")**, inheriting from **MOFState** (truncatable) and supporting the abstract interface derived for the association according to Rule (24). This valuetype contains one private member of type of the link-set's collection type (cp. Rule (25)), i.e. the sequence of derived link structs **<association_name>LinkSet**. The name for this member is **concatenate(format_2 (<association_name>), "_links")**.

Rule (27) If an association specializes another, this is mapped to inheritance of the abstract interfaces generated in Rule (24). The operations in the sub-association are only declared for those association ends that have a different name (for details see Section 7.6.3, “CMOF::Association,” on page 59).

The realization depends on the mapping profile (i.e., CCM or Base-IDL). If the association construct is mapped to IDL, the abstract interface definitions are supplemented with component definitions. If associations are not mapped, associations are only reflected by operations in the derived abstract instance interface and members in the valuetype (see Section 6.3.4, “Mapping of Tags,” on page 39.

Navigability and owned Properties

To make it unmistakable, we provide a short discussion about associations in EMOF, CMOF, and their relation to attributes with class type.

- **EMOF**: EMOF does not have associations, wherefore only properties with type “class” can be defined. If so, these attributes map to the usual operation definitions in the abstract interface (see Rule (11)-Rule (15)). If properties are defined to be the opposite of each other, the **set** operation of one property has also influence on the internal state of the opposite instance with respect to its reference knowledge.
- **CMOF**: If associations are mapped to IDL, then the only enhancement to the EMOF mapping is that there additionally exist a link-set. This link-set is also available through the component/interface derived for the association (Rule (24)) and is internally entangled with the access and update operations for the properties.

  In the (special) case that an association is mapped to IDL but is not navigable, neither in the one nor in the other direction, the instances have no knowledge about each other. Nevertheless, the link-set operations in the association interface can be used to “connect” the instances.

6.2.12 Mapping for Redefinitions

CMOF Redefinition of elements is always related to names or types of elements in a specialization relationship. Since most of the types in a model (classes, data types) are mapped to IDL interface types or valuetypes - which do not allow for overloading or redefinition directly - redefinition is rather a matter of operation signatures and their computational semantics.

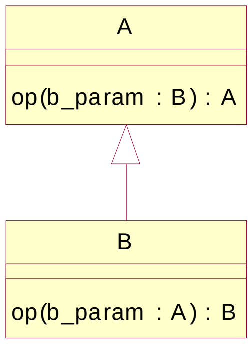
Rule (28) If a property redefines another property, then one of the following applies:

- If only the name is redefined by a property, new mutator operations are derived into the abstract interface with the redefined name as identifier in format_1 (according to Rule (12) or Rule (15)). The computational semantic of these newly declared operations is defined as a delegation to the redefined operations. The original operations should not be used and raise an exception when called in the more specific context (see Section 7.6.16, "CMOF::Redefinition," on page 62 for details).
- If only the type of a property is redefined, no additional operations are generated. Instead, the computational semantic of the operation (of Rule (12) or Rule (15)) is changed to return the more specific (redefined) type. A client can assume that he will receive the redefined type and thus can safely narrow the result.
- If both the name and the type are redefined, new update and access operations are derived with the new property's name as identifier (in format_1). Semantically, these replace the redefined operations and return the redefined (more specific) type. The "old" (inherited) operations should be made unavailable by raising an exception (see Section 7.6.16, "CMOF::Redefinition," on page 62 for more details).

Rule (29) Operations can be viewed in the same conceptual line. Therefore, the same as in Rule (28) applies for operations. Additionally, if a parameter type is redefined, the original definition of the operation is altered and receives the more general type as parameter type, but can expect (precondition) the specific type originally defined at invocation time. The reason for this is because conformance is required for redefinition (i.e., the parameter type may be altered in a contra-variant way).

Rule (30) Furthermore, if an enumeration is used as type for a property or an operation, the redefinition alters the original mapping of the redefined element. The enumeration type that is used instead is the type specified in the redefinition (the more specific enumeration). Since enumeration inheritance is realized through copying of all base-enum's literals, the operations remain consistent (see also Section 7.6.16, "CMOF::Redefinition," on page 62).

Example (2) Let **A**, **B** classes, **B** inheriting from **A**. Class **A** declares an operation **op** with return type **A** and a parameter of type **B**. Further, class **B** contains an operation that redefines the return type to the more specific type **B** and relaxes the parameter type to the more general type **A** (see figure below). For this scenario, the following IDL is derived:



```
abstract interface ACommon : MOFObject
{
    // the implementation of op can expect
    // that b_param is of type BCommon
    ACommon op ( in ACommon b_param );
};

abstract interface BCommon : ACommon
{
    // no operation is defined, but op must
    // return instances of type BCommon
};
```

6.2.13 Mapping of Reflection

Since the Reflective API overloads thin model interfaces with in most cases unneeded functionality, the proposed mapping drops the Reflective interface inheritance of MOF1.4. Instead, Reflection is regarded as MOF Service (orthogonal to any meta-model) and thus mapped as described in Section 6.2.21, “Support for the Reflection Service,” on page 34.

6.2.14 Mapping of Constraints

The definition of constraints in a metamodel and their monitoring in a repository is one of the computationally most important issues. The hitherto MOF, v1.4 mapping lacks a detailed specification (apart from sections 4.13.5 and 4.13.6 of [1]) and maps constraints to inconsiderably innocent string definitions.

In this specification, a different approach for MOF2.0 has been initiated. Since constraint evaluation is out of the scope of MOF, we suggest the MOF2.0 specification to make at least statements about the following constraint related issues (not only for OCL, but in a constraint language independent form):

- **Side-effects**. MOF2.0 should prescribe a constraint language to be absolutely side-effect free when used within metamodels. This has already been done in MOF, v1.4.
- **Evaluation policies**. Regardless of the language used, constraints can be classified in immediate constraints (or invariant) and deferred constraints (i.e., verifiable on demand). The MOF, v2.0 should make statements on how immediate constraints are handled with respect to an object’s life-cycle, since constraints essentially influence the computational semantic (e.g., not every invariant holds true directly after creating an object, because it may not have been initialized yet). Therefore, we suggest to include a few mandatory requirements:
    1. Dangling references are not considered when checking constraints. They are ignored.
    2. MOF Repositories (or at least the constraint objects themselves) must provide for a “switch” to toggle immediate constraints on. If once turned on, they remain checked throughout the lifetime of an object. Thus, for example, the creation of instances without parameters can safely be done and after several calls of initialization the immediate constraint checking can be turned on.
    3. The behavior of violating a constraint should be prescribed at least for the creation or deletion of objects. This is especially useful if the creation or deletion violates a constraint that is attached not directly to itself. For example,

if an instance is contained in a set that must contain a minimum number of objects, then an object of this set may not be deleted.

6.2.15 Mapping of other constructs

This section summarizes the mapping elements, (basically additional CMOF elements).

Rule (31) Instances of the classes Comment, Tag, Expression, and OpaqueExpression are mapped as follows:

- ***Comment***: if a comment is attached to an element of type Class, Association, or Package, a readonly attribute is derived into the common interface with identifier **concatenate ("comments_for_", <name_of_attached_element>)** of type **CommentList**:

  typedef sequence < wstring > CommentList;

  This sequence contains the Comment::body strings of all attached comments. If a comment is attached to any other element, only an IDL annotation is produced for that element.

- ***Tag***: Since tags are the MOF's extension mechanism, we will distinguish between the CCM and Base-IDL mapping. Refer to Section 6.2.20, "Mapping of Tags and the MOF Services," on page 33 and Section 6.3.4, "Mapping of Tags," on page 39.
- ***OpaqueExpression*** and ***Expression*** are not mapped to IDL, since neither the expression syntax is defined nor their semantic can be derived in the general case.
- ***Extents*** and ***URIExtents*** are not mapped to IDL.

The CCM Profile completes the Common mapping with component definitions, which support the hitherto derived IDL interfaces.

6.2.16 Mapping of Construct Class

Apart from the abstract interfaces and valuetypes derived for a class according to Rule (8) and Rule (9), component type definitions are generated, representing the class's instances.

Rule (32) If the class is not abstract, an IDL component definition is being generated in the same module as the abstract interface (cp. Rule (8)) with the name **format_1 (<class identifier>)**. The component is declared to support the generated abstract interface for the class.

Rule (33) The IDL component definition generated according to Rule (32) is extended by an attribute with the name **concatenate (format_2 (<class identifier>), "_state")**. The type of this attribute is the IDL valuetype generated according to Rule (9).

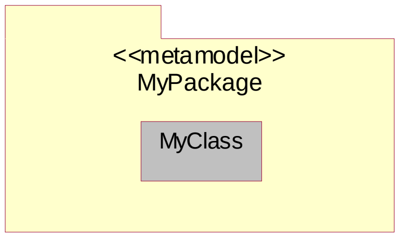
The attribute for the component's state can be used to access and update the component's state. The semantic of the attribute's update is more complicated and is discussed in detail in Section 7.7, "Merging of repository instances," on page 63.

In the MOF 1.4 IDL mapping, class proxy interfaces are generated, that are used to create class's instances. In this specification, this role is played by home interfaces generated for non abstract classes of a MOF model.

Rule (34) If the class in the model is not abstract, a home interface declaration for the component with the name **concatenate (format_1 (<class identifier>), "Home")** is being generated, managing the component generated following Rule (32).

Inheritance of classes is solely realized by the inheritance of the derived abstract interfaces (Section 6.2.7, "Mapping of Inheritance between Classes," on page 17). A component's support of its abstract interface assures type consistency.

Example (3) For a class **MyClass** defined in the package **MyPackage**, the following CORBA IDL definition is being generated:



```
module MyPackage
{

    // Common Mapping:
    abstract interface MyClassCommon : MOFObject
{ };
      valuetype MyClassState : truncatable MOFState
        supports MyClassCommon { };

    // CCM Mapping:
    component MyClass supports MyClassCommon {
            attribute MyClassState my_class_state;
    };
    home MyClassHome manages MyClass {
    };
};
```

6.2.17 Mapping of Attribute and Operation Constructs

While the mapping of attributes and operations for the common interface is already defined in Section 6.2.8, "Mapping of Attributes and Operation Constructs," on page 17, only the creation of instances remains to be specified.

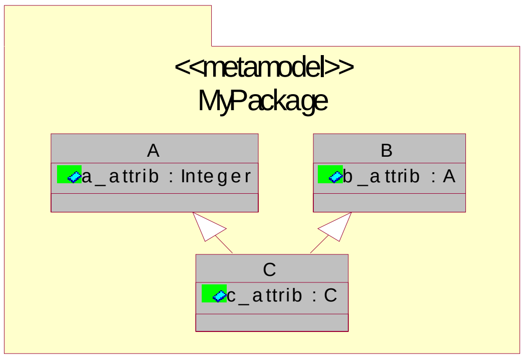
The home interface playing the role of a factory for a class's instances in a repository is being extended with two more factory operations: One that initializes the created instance with values for all attributes from parameters of the factory operation.

The second plays the role of a “copy constructor.” It initializes the created instance with values from an existing instance. It should be noted, that the type of the parameter of this operation is the abstract interface declared for a class in the model: This implies, that a user can either call the operation with the State valuetype generated according to Rule (9) or call it with a component reference of the same or more specific type.

Rule (35) If a class in the model contains non-derived attributes, the home interface declaration generated according to Rule (34) is extended with a factory operation named **concatenate ("create_and_init_", format_2 (<class identifier>))**, that defines for each of the attributes of that class and of all super classes a parameter for initialization.

Rule (36) In this case, a further factory operation named **concatenate ("copy_from_", format_2 (<class identifier>))** , that has one parameter of the type of the abstract interface generated for the class according to Rule (6).

Example (4) Let **A**, **B** and **C** be classes in the package **MyPackage**. **C** may inherit from **A** and **B**. **A** may have a public attribute **a_attrib** with multiplicity [1..1] of type **Integer**. **B** may have a public attribute **b_attrib** of type **A** with multiplicity [1..1] The class **C** may have a public attribute **c_attrib** of type **C** with multiplicity [1..1]. In this case, the following IDL definitions are being generated:



```
module MyPackage {
    abstract interface ACommon : MOFObject {
        long get_a_attrib ();
        void set_a_attrib ( in long a_attrib );
    };
   valuetype AState : truncatable MOFState
      supports ACommon {
        private long a_attrib;
    };
    component A supports ACommon {
        attribute AState a_state;
    } ;
    home AHome manages A {
        factory create_a ();
        factory create_and_init_a ( in long a_attrib );
        factory copy_from ( in ACommon the_a );
    };

    abstract interface BCommon : MOFObject {
        ACommon get_b_attrib ();
        void set_b_attrib ( in ACommon b_attrib );
    };
   valuetype BState : truncatable MOFState
      supports BCommon {
        private ACommon b_attrib;
    };
    component B supports BCommon {
        attribute BState b_state;
    };
    home BHome manages B {
        factory create_b ();
        factory create_and_init_b
            ( in long a_attrib,
              in ACommon b_attrib );
        factory copy_from ( in BCommon the_b );
    };
    abstract interface CCommon
      : ACommon, BCommon {
        CCommon get_c_attrib ();
        void set_c_attrib ( in CCommon c_attrib );
    };
```

```
                                                                valuetype CState : truncatable MOFState
                                                                   supports CCommon {
                                                                         private long a_attrib;
                                                                         private ACommon b_attrib;
                                                                         private CCommon c_attrib;
                                                                 };
                                                                 component C supports CCommon {
                                                                         attribute CState c_state;
                                                                 };
                                                                 home CHome manages C {
                                                                         factory create_c ();
                                                                         factory create_and_init_c
                                                                              ( in long a_attrib,
                                                                                in ACommon b_attrib,
                                                                                in CCommon c_attrib );
                                                                         factory copy_from ( in CCommon the_c );
                                                                  };
                                                            };
```

Shown below is how the generated IDL looks for an attribute with a multiplicity upper>1. The IDL generation rules presented in Section 6.2.4, "Mapping of Collections (MultiplicityElement)," on page 14 were used.

Example (5) Let's assume, the multiplicity for **b_attrib** of class **B** in Example (4) has the values upper>1, non-ordered and unique. According to Rule (15) , the following IDL would be generated:

```
module MyPackage {
    typedef sequence < ACommon > ASet;
    abstract interface ASetIterator : MOF::IteratorBase {
            ACommon get_value () raises (MofError);
            void insert_value ( in ACommon value ) raises (MofError);
            void modify_value( in ACommon value ) raises (MofError);

            ASetIterator next_one()raises (MofError);
            ASetIterator previous_one()raises (MofError);

            ASetIterator begin() raises (MofError);
            ASetIterator end() raises (MofError);
            boolean is_empty() raises (MofError);
        };
        valuetype ASetAsValue supports ASetIterator {
                    private ASet value;
        };
        interface ASetAsReference : ASetIterator {
            ASetAsValue as_value();
        };
    abstract interface BCommon {
            ASetIterator get_b_attrib ( );
            // modification is done by iterator operations
    };
};
```

Example (6) We continue with the Example (5), generating the IDL valuetype definition for **BState** as follows:

```
    module MyPackage {
        valuetype Bstate : truncatable MOFState supports BCommon {
                public ASetIteratorAsValue b_attrib;
        };
    };
```

6.2.18 Mapping of Associations

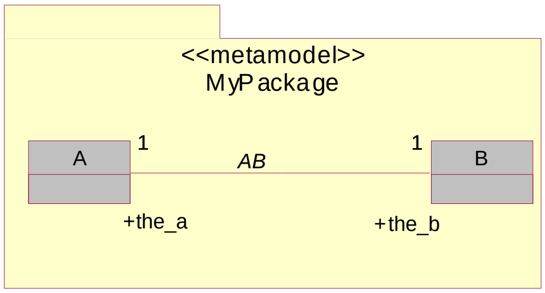
An association defined in a MOF model may or may not be mapped to CORBA IDL definitions. The mapping completes the definitions described in Section 6.2.11, "Mapping of Associations," on page 22 and Section 6.2.8, "Mapping of Attributes and Operation Constructs," on page 17.

Rule (37) If an association is mapped to IDL, a component declaration with the name **format_1 (<association identifier>)** is being generated, which declares to support the abstract interface produced by Rule (24).

For specification details, see Section 7.6.3, "CMOF::Association," on page 59.

Example (7) A model may contain the two classes **A** and **B** and an association AB between A and B with both ends having the multiplicity [1..1]. The association ends may be named **the_a** and **the_b**. In this example, the generated IDL looks as follows:

:



```
module MyPackage {

  abstract interface ACommon : MOFObject {};
  abstract interface BCommon : MOFObject {};

  struct ABLink {
     ACommon the_a;
     BCommon the_b;
  };
  typedef sequence < ABLink > ABLinkSet;

  abstract interface ABCommon : MOFObject {
     ABLinkSetIterator all_ab_links() raises
(MofError);
     void create_link_in_ab( in ABLink new_link )
          raises  (MofError);
     boolean link_exists( in ABLink link ) raises
(MofError);
     void remove_link( in ABLink link )
          raises (MofError, NotFound);
     BCommon linked_objects_the_a( in A the_a )
          raises (MofError);
     ACommon linked_objects_the_b( in B the_b )
          raises (MofError);
  };

  valuetype ABState : truncatable MOFState
          supports ABCommon {
     private ABLinkSet ab_links;
  };

  component AB supports ABCommon {
     attribute ABState ab_state;
  };

};
```

6.2.19 Mapping of Package Constructs

In MOF 1.4, packages were mapped to IDL interfaces, containing **readonly** attributes for the class proxy, association and package interfaces that are contained by this package.

In the approach we've taken here, packages will be mapped to abstract interfaces, components, and home definitions.

Rule (38) A package definition in a MOF model is mapped to an IDL abstract interface definition with the name **concatenate (format_1 (< package identifier>), "Package")**. For each contained class an operation is defined within this abstract interface named **format_2 (<class identifier>)** with the return type **concatenate (format_1 (<class identifier>), "Home")**. The operations have no parameters and return the corresponding home interfaces for the classes. Furthermore, the package's abstract interface contains operations for each owned package or association with name **format_2(<package or association identifier>)**. The return type of these operations is the corresponding abstract interface derived for a package or association (see Rule (24)).

Rule (39) Furthermore, a component with the name **format_1 (< package identifier>)** is generated supporting the produced abstract interface. If the package is the outermost package (in case of package merge this is the result of the merge), a home definition is produced named **concatenate (format_1 (< package identifier>), "Home")** that manages the component.

NOTE: It should be noted, that there's no PackageFactory being produced. This role is played by the corresponding home.

Example (9) For the package MyPackage in Example (4), the following IDL definitions are produced:

```
module  MyPackage {
    component A supports ACommon { /* … */ };
    component B supports BCommon { /* … */ };
    component C supports CCommon { /* … */ };
    home AHome manages A { /* … */ };
    home BHome manages B {/* … */ };
    home CHome manages C {/* … */ };
    abstract interface MyPackagePackage {
            AHome a();
            BHome b();
            CHome c();
    };

    component MyPackage
            supports MyPackagePackage { };
    home MyPackageHome
            manages MyPackage { };
};
```

6.2.20 Mapping of Tags and the MOF Services

As stated earlier, services such as reflection or repository notifications are mapped onto behavioral features of the generated components. The interfaces generated for elements of a MOF model do not inherit base interfaces from packages defined for these services.

This specification reuses the concept of standard tags for the IDL mapping as defined in MOF, v1.4 (section 5.6). Since tags are the MOF's extension mechanism, the CCM mapping must include tag definitions for the MOFServices:

Rule (40) Standardized tags for the IDL mapping define standardized service interfaces. These interfaces are supported through facets at the component (see 8.3.6 as example).

6.2.21 Support for the Reflection Service

This specification redefines the interfaces of the Reflective module that was created to support the Reflection service. Thereby, including the reflection mechanism serves as a comprehensive example, how services are integrated into the derived components via facets (details on the reflective interfaces itself can be found in chapter 8).

Rule (41) If a class/package/association in a MOF model shall support the Reflective API, a facet named **reflective** is added to the component definition for that class. The type of that facet is:

- **EMOF**:
  - **EMOF::CCMReflective::RefCCMObject** in case of a class,
  - **EMOF::CCMReflective::RefCCMBaseObject** in case of a package.
- **CMOF**:
  - **CMOF::CCMReflective::RefCCMObject** in case of a class,
  - **CMOF::CCMReflective::RefCCMAssociation** in case of an association, and
  - **CMOF::CCMReflective::RefCCMBaseObject** in case of a package.

The facet for the Reflection service is always being named **reflective**. Since the generated components do not inherit from other components, this will not lead to name clashes.

Rule (42) Furthermore, every derived home interface also supports the reflective interface **CCMReflective::RefCCMHome** (contained in the EMOF or CMOF modules respectively).

Example (10) If the class C and the package MyPackage of Example (10) shall support Reflection, the following IDL definitions will be generated (as CMOF model):

```
module  MyPackage {

        component A supports ACommon {
            attribute AState a_state;
            provides CMOF::CCMReflective::RefCCMObject reflective;
    };

    home AHome
            supports CMOF::CCMReflective::RefCCMHome
            manages A
    {
      //...
    };

    abstract interface MyPackagePackage {
            AHome a();
            BHome b();
            CHome c();
    };

    component MyPackage supports MyPackagePackage {
        provides CMOF::CCMReflective::RefBaseObject reflective;
    };

    home MyPackageHome
            supports CMOF::CCMReflective::RefCCMHome
            manages MyPackage
    {
      //...
    };
};
```

6.2.22 Support for Notification based Model Change Communication

The currently (MOF 1.4) generated IDL definitions imply, that a generated repository never communicates changes to repository objects (i.e., model elements) actively. Instead, a repository client always actively queries the repository to get informed about model changes. We propose to add a new service, which we call Active Repository that can be used to initiate the communication of model changes actively by the repository. For this purpose, we declare event ports in the scope of those components that should support this service. The event type is implied by the state valuetype defined for a class according to Rule (4).

Rule (43)	If a class of a MOF model shall support the Active Repository service, a **publishes** declaration is added to the component definition of that class. The name of this declaration is **concatenate ("changes_", format_2 (<class identifier>))**.
Rule (44)	The type of the publishes declaration is an **eventtype** named **concatenate (format_1 (<class identifier>), "Changes")**. This eventtype inherits from the state valuetype for the class produced according to Rule (1).

For specification details, see Chapter 7.

Example (11) Assume, that the class **C** of Example (4) shall support both the Reflection and Active Repository services. According to the rules for MOF services mapping, the following IDL would be produced:

```
module MyPackage {

    /* definitions for class C */
    abstract interface CCommon : ACommon, BCommon {
        CCommon get_c_attrib ();
        void set_c_attrib ( in CCommon c_attrib );
    };
    valuetype CState supports CCommon {
        private long a_attrib;
        private ACommon b_attrib;
        private CCommon c_attrib;
    };

    eventtype CChanges : CState {};

    component C supports CCommon {
        attribute CState c_state;
        provides Reflective::RefObject reflective;
        publishes CChanges changes_c;
    };

    home CHome manages C { /* ... */ };
};
```

6.2.23 Querying CCM models

In MOF1.4, querying of metamodel instances in a repository was realized by the **all_of_type** and **all_of_class** operations of the class proxy interface. Intuitively, in CCM the finder operations of a home should be reused for the purpose of querying a container's content. Unfortunately, finders do not provide to query a container for all instances of a specific component type nor allow the returning of collections of components at all.

Rule (45) For every class in a metamodel, two operations with identifier **all_of_type** and **all_of_class** are declared within the home that is generated according to Rule (34). Both operations obtain one **in** parameter: **boolean as_value** and a return type of the (**Set**) iterator interface that is derived in Rule (3). The semantic is the same as for the **all_of_*** operations in MOF1.4.

Another suggestion for a more advanced query-interface would be to reuse the concept of EJB to define finders. The idea is to supply query-requests as parameters in kind of EJB internal QueryLanguage (EJB QL) like style. Thus, simply one operation is defined that directly supports “words” of a query language (in opposite to EJB, where only predefined queries are possible). An example for OCL would look like:

<AbstractSetIterator> ocl_query (in string query_expression, in boolean result_as_value);

Thus, a client is able to specify OCL expressions for an explicit search via any combination of characteristics of the instances. For example, a query string like **self.name = 'HardToImplement'** returns an iterator to all instances with an attribute called **name** set to **'HardToImplement'**.

6.3 Base-IDL Mapping

Apart from the component mapping, this section provides the opportunity of a true Base-IDL mapping to cope with non-component environments and MOF1.4 clients. It must be stressed that this mapping is an alternative mapping to CCM - since several names and interfaces collide with definitions for the CCM. and we do not believe that the equivalent IDL suffices usability needs. Moreover, the authors prefer the CCM mapping approach.

6.3.1 Mapping of Packages

The package mapping is similar to MOF, v1.4. Mainly two interfaces are derived, one as a factory and one representing the package's instances. Nevertheless, this structure is extended by an additional interface for query operations for models and the lifting of functionality of class proxy interfaces into the package instance interface. Class proxy interfaces are dropped (see Section 6.3.2, "Mapping of Construct Class," on page 37):

Rule (46) For each package, an IDL interface definition is being generated - taking into account 8.2.5 - with identifier **concatenate (format_1 (<package identifier>), "Factory")**. This interface is identical to that from section 5.8.3 of the MOF1.4 specification and contains the same operations.

Rule (47) Additionally, a package instance interface is derived according to section 5.8.4. of the MOF1.4 specification with name **format_1 (<package identifier>)**, but without attributes for classes and associations.

Rule (48) Furthermore, an interface with name **concatenate (format_1 (<package identifier>), "Query")** is derived (referred to as the package's Query Interface). This interface contains operations for querying a model's elements contained by the package. Additionally, the package instance interface is enhanced with a readonly attribute with type of the query interface.

Operations and attributes for contained classes and associations are specified in 6.4.2, 6.4.4, and 6.4.3, respectively.

Conceptually, the package instance interface is dedicated to create instances of contained model elements and manage their life-cycle, whereas the Query interface is utilized for analysis and exploration of the models.

6.3.2 Mapping of Construct Class

The representation of M1-level instances is achieved through the derivation of a non-abstract interface, apart from the abstract interfaces and valuetypes derived for a class according to Rule (8) and Rule (9).

Rule (49) If the class is not abstract, an IDL interface definition is being generated in the same module as the abstract interface (cp. Rule (8)) with the name **format_1 (<class identifier>)**. This interface is declared to inherit from the generated common (abstract) interface for the class and from all other base interfaces (non-abstract) for that class in terms of service definitions.

Rule (50) To access the instances state, an additional operation is generated into the non-abstract interface of Rule (49) with identifier **concatenate ("get_", concatenate (format_2(<class_name>) , "_state"))** typed to the valuetype generated according to Rule (9).

The get-operation can be used to access the instance's state (update is realized by the copy_from). The semantic of the attribute's update is not trivial and is discussed in Section 7.7, "Merging of repository instances," on page 63.

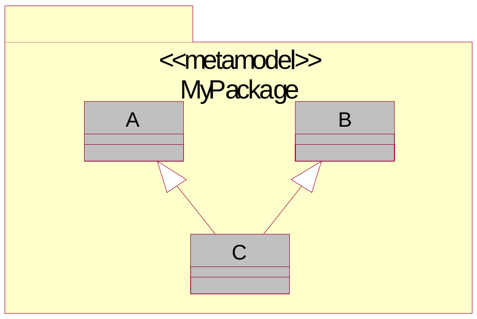
In the MOF, v1.4 IDL mapping, class proxy interfaces are generated, that are used to create class's instances. In this specification, we drop the class proxy interface and generate the creation operation(s) for instances into the package interface that is derived according to Rule (47).

Rule (51) If the class in the model is not abstract, a factory operation to create instances without parameters with the name **concatenate ("create_", format_2 (<class identifier>))** is generated. Additionally, a **copy_from** and a **create_and_init** is operation is derived analogous to Rule (35) and Rule (36).

It should be noted, that inheritance of classes is realized by the inheritance of the derived abstract interfaces (cp. 8.2.7) and additionally via the instance interfaces.

For specification details, see Chapter 7.

Example (12) Let **A**, **B** and **C** be classes in the package **MyPackage**. **C** may inherit from **A** and **B**. In this case, the following IDL definitions are being generated:



```
module MyPackage {
    // Common Mapping:
    abstract interface ACommon : MOFObject { };
    abstract interface BCommon : MOFObject { };
    abstract interface CCommon
      : ACommon, BCommon { };
    valuetype AState : truncatable MOFState
      supports ACommon { };
    valuetype BState : truncatable MOFState
      supports BCommon { };
    valuetype CState : truncatable MOFState
      supports CCommon { };

    // Base-IDL Mapping:
    interface MyPackageFactory { /* ... */ };

    interface MyPackage  {
        A create_a();
        B create_b();
        C create_c();
    };
    interface A : ACommon { };
    interface B : BCommon { };
    interface C : CCommon, A, B { };
} ;
```

6.3.3 Mapping of Attributes and Operations

The mapping of attributes and operations is specified in Section 6.2.8, “Mapping of Attributes and Operation Constructs,” on page 17. This section adds only the create operations for the instances within the package interface.

The package interface is being extended with two more create operations for each class (cp. 6.3.2). The first features initialization of all attributes of the created instance, the second plays the role of a "copy constructor": it initializes the created instance with values from an existing instance. As in the CCM mapping, the type of its copy-parameter is that of the abstract interface (Rule (8)) to allow initialization by an object reference or by a valuetype.

Rule (52) If a class in the model contains non-derived attributes, the containing package interface declaration generated according to Rule (47) is extended with a factory operation named **concatenate ("create_and_init_", format_2 (<class identifier>))**, that defines for each of the attributes of that class and of all super classes a parameter for initialization.

In this case, a further factory operation named **concatenate ("copy_from_", format_2 (<class identifier>))**, that has one parameter of the type of the abstract interface generated for the class according to Rule (6).

6.3.4 Mapping of Tags

Tags are mapped similar to MOF1.4. As described in Section 6.3.5 for the CCM mapping, only standardized IDL tags lead to IDL mapping altering.

Rule (53) The concept of tags is reused from MOF1.4, but with the modification that the non-abstract interfaces derived according to Rule (49) are used for inheritance issues. See 6.3.6, 'Support of MOF Services and Reflection' for an example.

6.3.5 Mapping of Associations and References

Apart from the derivation for associations in the Common Mapping as described in Section 6.2, "Common Mapping Rules," on page 12, one more interface is derived:

Rule (54) If an association is mapped to IDL, an interface with identifier **format_1 (<assocation_name>)** is generated inheriting from the abstract interface derived in Rule (24).

As can be seen, this is the same concept as for classes.

6.3.6 Support of MOF Services and Reflection

Since MOF Services provide multiple views for model elements, their support in IDL is realized by inheritance of functionality into the instance interfaces. Thereby, these base interfaces of the service may be generated or not, depending on service.

An example for non-generated interfaces is the reflective API.

Rule (55) If a class/package/association in a MOF model shall support the Reflective API, an inheritance relation is added to the instance interface definition for that class as for every other service. The type of the reflective interface is:

- **EMOF**:
  - **EMOF::Reflective::RefObject** in case of a class,
  - **EMOF::Reflective::RefPackage** and **EMOF::Reflective::RefFactory** in case of a package.
- **CMOF**:
  - **CMOF::Reflective::RefObject** in case of a class,
  - **CMOF::Reflective::RefAssociation** in case of an association and
  - **CMOF::Reflective::RefPackage** and **CMOF::Reflective::RefFactory** in case of a package.

6.3.7 Support for Notification based Model Change Communication

A detailed specification of a Notification Service for the Base-IDL mapping will not be provided in this specification.

7 Specification Details

This chapter contains the language mapping details for the specification provided in Chapter 5. A more formal definition of the mapping rules is done in the convenience document [13] using OCL invariants to express the transformation. In the following, the computational semantics is described and details of the generated constructs are outlined in Chapter 6.

7.1 Computational Semantics

Most of the computational semantics of the former MOF1.4 to IDL specification ([1]) is reused in this specification, with modifications considering the MOF2 abstract semantics. Comparing the IDL constructs derived in the MOF1.4-to-IDL mapping to this specification, the similarities of operations for access on the models are obvious. What essentially changed can be outlined as follows:

- Derived interfaces are now derived as abstract interfaces. Thus, the call semantic is enhanced significantly, but has only small influence on the "real" computational semantic. Most of the set, get, create, etc. operations maintain their functionality.
- The object management of a MOF repository has been improved, since most of the CORBA 3.0 built-in features of components and homes have been utilized to represent metamodel instances. Therefore, the "flat" element-to-interface mapping was restructured and simplified from the user perspective, because CCM concepts that provide exactly these facilities are used instead. Moreover, most of the MOF computational semantic can be directly represented by those CCM build-in concepts. For Base-IDL as alternative, the common features of MOF1.4 are simply updated to a more advanced mapping.

Thus, the following subsections will focus on the additions to and refinements of the MOF2 abstract semantic. Since MOF2 imports most of the semantic concepts from the UML2 Infrastructure which defines them in depth, we will in most cases give a rather short explanation. Ex ante, the general concepts for the computational semantic will be detailed:

- ***Call semantic***. Switching the call semantic from references to valuetypes should not affect the operations' general semantical task. With the addition of valuetypes, basically two cases can be distinguished:

1. *Querying of instances*. The query (or "get") operations remain computationally unchanged. If an object is configured to return valuetypes, then this cannot affect the operations behavior. The only difference is the underlying call mediation of the returned result (i.e., changes to the valuetypes remain local).
2. *Updating of instances*. The update (or "set") operations obtain a more complex semantic. A client must be aware that changes to a previously returned valuetype remain local and are not automatically propagated back to the repository. Instead, this propagation must be done in a separate call to update the repository instances state with the locally modified state. 7.7, 'Merging of repository instances' covers this issue in depth.

Thus, returning a reference or valuetype will be configured through the **MOFObject::set_call_semantic operation** (see Section 7.3, "Global Definitions for EMOF and CMOF," on page 44).

- ***Packages and Extents***. The computational model for instances in a repository is based on package extents (as defined in MOF1.4) **and** unique identifiers. The following subsections will abstract from terms of extents (and aspects of e.g., closure rules) and will assume the existence of only <u>one metamodel instance</u> at time of execution. This approach enables a short description of the computational tasks, which can be easily extended afterwards with the common known extent rules of MOF1.4. Note that if a CCM container manages multiple metamodel instances at a time, this has implications on, for example, the usage of home finders. The implementation must guarantee that derived operations return appropriate homes that correspond to the accessed extent[1].

Anyhow, we associate each object with a unique identifier (UUID) that is different for every existing object (at least in the context of a repository server). For our computational model, a unique identifier within an extent is not sufficient: clients may modify copies of instances outside the repository (valuetypes) and propagate them back. Since we need some mechanism to uniquely identify an object with respect to figure out, whether an object is already in an extent or not (a user may, for example, transfer objects from one extent via valuetype copies into another extent). Thus, we refer to the unique id of each object as *MofId* and use it as computational model.

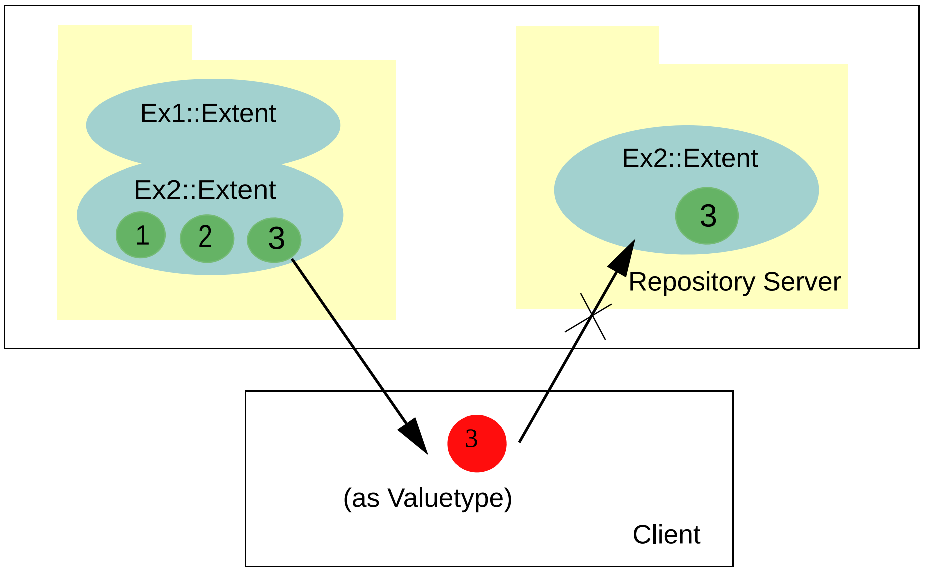
Moreover, while the MofId describes an object uniquely, we do neither prescribe to use e.g. primary keys for CCM components nor any other mechanisms (For example, the derived components may be implemented using process components).

Note for clarification of the notion of Extents in the MOF-IDL mapping.

Since the meaning of Extents changed a lot since MOF, v1.4, we should clarify our notion of a mapping of extents regarding a realization in a CORBA environment. Basically, we do not define concrete IDL operations/interfaces for extents (initially) for two reasons:

(1) A concrete realization heavily depends on the “interference” of extents with the Facility spec concepts. While watching the UML/MOF 2.0 FTF fruitful discussions and issues regarding XMI, we found it best to defer the alignment of extents mapped to IDL to the FTF process for the MOF2.0-IDL according to actual results. This is especially true for URIExtents.

(2) Due to the reference/value call-semantic choices. The following picture visualizes the dilemma:



1. A vendor may choose to resolve extent-related issues via the repository ids.

Note (continued)	If a client receives the state of an object as IDL valuetype, it may transmit it to a different extent where it may clash with an object that has the same identifier. One possible occurrence of this special case may be the transition between two versions of a model. It remains still open, whether extents can be cross cutting through different versions, branches, etc. However, we truly believe that an FTF is able to align and refine the mapping for extents/ URIExtents with upcoming issues within the finalization process.

- ***MOF Services*** may be supported by elements in a repository or not. If support is enabled, the service may alter the execution semantic of the derived operations significantly. For example, let an object support transactions, then surely this has influence on the availability and update semantic of the object. Nevertheless, services should alterations explicit.
- ***Services and repository root reference***. As in MOF, v1.4, the specification of an *initial interface* for a repository implementing derived IDL may be included in the specification for the purposes of accessing configuration information of service support. Thus, clients can query available services in the same style as the **ORB::resolve_initial_references** operation:

```
const string SERVICE_REFLECTION = “MofReflection”;
const string SERVICE_TRANSACTION = “MofTransactions”;
const string SERVICE_VERSIONING = “MofVersioning”;
// …

typedef sequence < string > ServiceList;

interface MofRepositoryRoot {
    ServiceList resolve_mof_services();

    // Possibly further “root” operations for
    // accessing repository capabilities
};
```

The **resolve_mof_services** operation returns a list of all services supported by the metamodel instances.

7.2 PrimitiveTypes Mapping

The primitive types as defined in *InfrastructureLibrary::Core* are mapped as follows:

PrimitiveType instance	Corresponding IDL type
Boolean	boolean
String	wstring
Integer	long
UnlimitedNatural	unsigned long

Furthermore, we provide default values for the primitive types. These default values are used if, for example, a default constructor is used to initialize an object:

PrimitiveType instance	IDL type default value
Boolean	false
String	„“
Integer	0
UnlimitedNatural	0

7.3 Global Definitions for EMOF and CMOF

The global definitions apply to all parts of the mapping, regardless whether a model is compliant to EMOF or CMOF. These IDL definitions are not metamodel specific and provide some basic capabilities for the object lifecycle.

```
module MOF {

  exception MofError;
  typedef MofId wstring;

  abstract interface MOFObject {
     MofId get_mof_id () raises (MofError);
     boolean is_equal ( in MOFObject other_object )
            raises (MofError);
     void delete () raises (MofError);
     MofErrorBagIterator verify( in unsigned long depth )
            raises (MofError);

     void set_value_depth( in unsigned long depth )
            raises (MofError);
     unsigned long get_value_depth ()
            raises (MofError);

     void set_call_semantic( in boolean as_value )
            raises (MofError);
     boolean get_call_semantic ()
            raises (MofError);
  };

  valuetype MOFState supports MOFObject {
     private MOFObject origin;
  };
};
```

MOFObject

The MOFObject interface serves as the base interface for all derived interfaces. In general, a MOFObject represents an instance of a metamodel class.

MOFObject::get_mof_id

Parameters

(none)

Semantics

The **get_mof_id** returns the identifier that uniquely identifies an object (UUID, not simply an identifier within an extent, cp. 9.1). By default, object identity is established when creating an object and can not be changed during the objects lifetime.

Nevertheless, the objects identity to a client is **not** determined by this identifier; it is rather needed for managing the object within a repository. Since MOF2 allows establishing identity through owned properties which have an **isID** flag set to true, the **is_equal** operation must be used to check whether two objects are identical.

In the context of a valuetype, this operation must return the identifier for the object in the repository that was the origin for its state, that is:

```
origin.get_mof_id()
```

Thus, the valuetype remains an image of an objects state with a back-reference to its original object. Note that this mechanism is used later on for the merging of objects as described in Section 7.7, “Merging of repository instances,” on page 63.

Returns

MofId is a stringified representation of the identifier for this object. The format for this **wstring** is open to implementors.

Exception

MofError is raised if the object in the repository is not available.

MOFObject::is_equal

Parameters

MOFObject other_object : reference to the object where the objects identity is checked against.

Semantics

The operation checks, whether the object that is passed as parameter is the same as the object on which the operation is invoked. The operation has to do more than simply compare the MofIds, because identity is not always creation based.

Derived interfaces override this operation to implement the specific identity semantic (with respect to an ***isID*** flag of an owned property) or other equivalence semantic.

If the operation is invoked on a valuetype, it must return true if the valuetype is a copy of the element passed as parameter (i.e., in case of the absence of an ***isID*** flag):

```
origin.get_mof_id() == other_object.get_mof_id();
```

Returns

true, if the objects are the same in terms of identity, false otherwise.

Exception

MofError is raised if an object in the repository is not available.

MOFObject::delete

Parameters

(none)

Semantics

Removes (i.e., destroys) an object in the repository. If the object directly contains others, these are also deleted (transitively). The semantic is the same as the **refDelete** operation in MOF1.4.

Note, that references to the object are not removed automatically. This may result in so called *dangling-references* which still refer to this object. It is up to a client to restore the repository into a valid state.

Returns

(none)

Exception

If the object cannot be destroyed (e.g., because a constraint is violated in the case of deletion or the object has been removed by another client in the meantime), the MofError exception is raised with **INVALID_DELETION_VIOLATION** set.

MOFObject::set_value_depth

Parameters

unsigned long depth: determines the depth of valuetype assembly

Semantics

This operation determines the access semantics for the state members that are derived into the component/interfaces for a specific model element (cp. e.g., Class mapping). It determines how deep the returned valuetype will contain other valuetypes for associated elements (associated via properties of classifier type). Thereby two cases can be differentiated:

1. valuetypes can contain the specific interface/component inheriting from/supporting an abstract interface (depending on the CCM/Base-IDL mapping) by *reference*, or
2. valuetypes can contain the other elements in turn as *valuetypes*, since these are also supporting the abstract interfaces derived for an element.

The **set_value_depth** operation controls this semantic and determines for the second case, how deep the valuetypes are packaged. The **depth** parameter can be viewed as a kind of “hop-count” which states, how many valuetypes are included counting each referenced element with the value of one:

- **depth = 0 or 1**: every operation that is typed to an abstract “Common” interface returns a reference to the specific type in the respective mapping (Component reference for CCM mapping or non-abstract interface for Base-IDL mapping).
- **depth > 1**: every operation that is typed to an abstract “Common” interface returns corresponding valuetypes for

depth - 1 reachable instances (i.e. those that are reachable via class-typed properties). For example, a **depth** value of one means: return only the valuetype itself and all other “Common” typed attributes as CORBA object reference; a **depth** value of two means, that the valuetype itself will contain valuetypes for the immediate neighbors and then references to neighbors in turn of these instances and so on.

Note that this operation controls the access for each instance separately and ignores the setting of associated instances depth value (i.e., the **depth** value of the object that receives the operation-call is dominant). The assembly of these nested valuetypes that are built via this “packaging” step is left to the implementor of the repository (intelligent *sharing* of identical valuetype instances through referencing or duplication).

Furthermore, repositories may choose to disable this feature (object references are always returned). Initially, the value assumed at creation time for an object is zero.

If the object has no references to other objects via properties, the depth parameter has no effect. Setting **depth** value to a higher number than the reachable number of instances will result in multiple assembly of instances. If properties reference back to the origin object, then this object may be contained twice (or even more often).

Returns

(none)

Exception

MofError is raised if the object is not available.

MOFObject::get_value_depth

Parameters

(none)

Semantics

Returns the actual value for the packaging/assembly of valuetypes. The initial value for the value packaging is zero (i.e., object references are returned).

Returns

Number of valuetype instances that will be received within a valuetype when calling an operation that is derived for a property with classifier type.

Exception

MofError is raised if the object is not available.

MOFObject::set_call_semantic

Parameters

boolean as_value: determines whether a reference or a valuetype is returned.

Semantics

This operation determines the access semantics for operations that are derived for properties of classifier type within the abstract interfaces for a specific model element. It determines what is returned when an operation is invoked:

1. the operations can return the specific interface/component inheriting from/supporting an abstract interface (depending on the CCM/Base-IDL mapping) as *object reference*, or

2. the operations can return the referenced elements as *valuetypes*, since these are also supporting the abstract interfaces derived for an element.

This is similar to the **set_value_depth** operation, but with the difference, that there is no deep packaging of valuetypes.

Initially, the default value at creation time is false (means: return references).

Returns

(none)

Exception

MofError is raised if the object is not available.

MOFObject::get_call_semantic

Parameters

(none)

Semantics

Returns the configured call semantic for the object.

Returns

true, if the object returns valuetypes for all operations derived for properties, false otherwise.

Exception

MofError is raised if the object is not available.

MOFState

The **MOFState** valuetype serves as a base for all valuetypes derived for a specific metamodel.

MOFState::origin

Semantics

This private member holds a (back-) reference to the object in the repository from which it is a state-copy. It is used e.g., if the **get_mof_id** must access the original object in the **is_equal** operation (see *MOFObject::is_equal*).

The member is private, because a client must not alter this information.

7.4 Exceptions

As already explained in Section 6.2.10, “Mapping of Exceptions,” on page 20, we intend to reuse the MOF, v1.4 exceptions in general with some modifications. The resulting IDL definitions after modification are:

const string UNDERFLOW_VIOLATION = "org.omg.mof:structural.underflow";

```
const string OVERFLOW_VIOLATION = "org.omg.mof:structural.overflow";
const string DUPLICATE_VIOLATION = "org.omg.mof:structural.duplicate";
const string REFERENCE_CLOSURE_VIOLATION = "org.omg.mof:structural.reference_closure";
const string SUPERTYPE_CLOSURE_VIOLATION = "org.omg.mof:structural.supertype_closure";
const string COMPOSITION_CYCLE_VIOLATION = "org.omg.mof:structural.composition_cycle";
const string COMPOSITION_CLOSURE_VIOLATION = "org.omg.mof:structural.composition_closure";
const string INVALID_OBJECT_VIOLATION = "org.omg.mof:structural.invalid_object";
const string NIL_OBJECT_VIOLATION = "org.omg.mof:structural.nil_object";
const string INACCESSIBLE_OBJECT_VIOLATION = "org.omg.mof:structural.inaccessible_object";
const string ALREADY_EXISTS_VIOLATION = "org.omg.mof:structural.already_exists";
const string INVALID_DESIGNATOR_VIOLATION = "org.omg.mof:reflective.invalid_designator";
const string WRONG_DESIGNATOR_VIOLATION = "org.omg.mof:reflective.wrong_designator_kind";
const string UNKNOWN_DESIGNATOR_VIOLATION = "org.omg.mof:reflective.unknown_designator";
const string ABSTRACT_CLASS_VIOLATION = "org.omg.mof:reflective.abstract_class";
const string NOT_CHANGEABLE_VIOLATION = "org.omg.mof:reflective.not_changeable";
const string NOT_NAVIGABLE_VIOLATION = "org.omg.mof:reflective.not_navigable";
const string NOT_PUBLIC_VIOLATION = "org.omg.mof:reflective.not_public";
const string WRONG_SCOPE_VIOLATION = "org.omg.mof:reflective.wrong_scope";
const string WRONG_MULTIPLICITY_VIOLATION = "org.omg.mof:reflective.wrong_multiplicity";
const string WRONG_TYPE_VIOLATION = "org.omg.mof:reflective.wrong_type";
const string WRONG_NUMBER_PARAMETERS_VIOLATION = "org.omg.mof:reflective.wrong_number_parameters";
const string INVALID_DELETION_VIOLATION = "org.omg.mof:reflective.invalid_deletion";

const string ILLEGAL_ARGUMENT_VIOLATION = "org.omg.mof:reflective.illegal_argument";
const string ADD_LINK_TO_UNION_VIOLATION = "org.omg.mof:reflective.add_link_to_union_violation";
const string REMOVE_LINK_FROM_UNION_VIOLATION =
"org.omg.mof:reflective.remove_link_from_union_violation";

struct NamedValueType {
    wstring name;
    any value;
};

typedef sequence < NamedValueType > NamedValueList;

exception MofError {
    string error_kind;
    MOFObject element_in_error;
    NamedValueList extra_info;
    wstring error_description;
};

exception NotSet {};
exception NotFound {};
```

7.5 Mapping for EMOF Compliant Models

EMOF compliance is achieved by using only instances of classes of the MOF::EMOF package. The following subsections provide the CCM mapping for these elements specified (for a discussion and explanations refer to Chapter 8). For a more formal specification refer to the OCL constraints in the convenience document [13].

7.5.1 EMOF::Package

An instance of EMOF::Package is mapped to a module definition and components/interfaces to create instances of this package. Containment is consistently transformed to modules.

Derived Package Component

The component derived for a package represents an instance of the package; this corresponds to package instance interface in MOF1.4. The supported abstract interface contains operations for accessing all homes of contained elements. Note that the home finder of CCM may not be used without restrictions, since multiple metamodel instances may be managed by a container. Therefore, the package interface must provide navigation methods to homes for all contained elements.

Derived Package Home interface

Manages package components. Corresponds to package factory in MOF1.4. Note that the usage of home finders may be restricted due to the issues mentioned above.

<package_name>Home::create

Parameters

(none)

Semantics

Creates an instance of the component. Corresponds to create_package operation in MOF1.4. All nested package components are created at the same time.

Returns

Reference to the new component if creation was successful.

Exception

MofError is raised if an error occurred.

7.5.2 EMOF::Class

Classes are mapped to an abstract interface, a valuetype, a component, and a home definition.

Derived Class Component

A component instance represents a class instance. It supports the abstract interface which contains mutator operations for the components state. Component together with abstract interface correspond to the instance interface in MOF, v1.4.

Derived Home interface

The Home manages the class component and corresponds to the class proxy interface of MOF, v1.4. It does not provide the *all_of_** operations of the MOF, v1.4 proxy interface (see discussion in Section 6.3.8, “Querying CCM models,” on page 33).

<class_name>Home::create

Parameters

(none)

Semantics

Parameterless create operation to create an instance of the component with default initialization. Attributes of primitive type are initialized using the default values specified in 7.2. See also the discussion for immediate constraints in Section 6.2.14, "Mapping of Constraints," on page 25 and default values in Section 6.2.8, "Mapping of Attributes and Operation Constructs," on page 17. There exists no corresponding operation in MOF, v1.4.

Returns

Component instance if creation was successful.

Exception

MofError is raised if an error occurred. Violation of constraints may also influence the creation process and may also lead to an exception.

<class_name>Home::create_and_init

Parameters

in parameters for all non-derived attributes (incl. superclasses).

Semantics

Create operation with member initialization to create an instance with initialization. Corresponds to *create_<class_name>* operation of class proxy interface in MOF, v1.4.

Returns

Component instance if creation and initialization was successful.

Exception

If an error occurred, **MofError** is raised. Note that violation of constraints may lead to exceptions.

<class_name>Home::copy_from

Parameters

in *<class_abstract_interface>* : instance to copy the state from.

Semantics

Creates an instance and initializes attributes with those of the specified parameter. No corresponding operation in MOF, v1.4. See also Section 7.7, "Merging of repository instances," on page 63 for merging related issues.

Returns

Component instance with the state of the parameter.

Exception

MofError is raised if an error occurs.

7.5.3 EMOF::Property

Properties are mapped to mutator operations into the abstract interface derived for the owning class.

If a property's multiplicity upper = 1, then the following get-operation is derived:

<class_name>Common::get_<prop_name>

Parameters

(none)

Semantics

If the property's type is primitive, the value is returned. If it is a class, the **MOFObject::set_call_semantic** operation determines whether the referenced instance is returned as CORBA object reference or as valuetype. Corresponds to an attribute's *get*-operation in MOF, v1.4.

Returns

The property's value.

Exception

MofError is raised if an error occurs. This should not arise as result of a constraint violation. Raises **NotSet**, if the attribute is optional and not set.

If the property's isReadOnly attribute is false and the multiplicity upper = 1, then the following set-operation is derived:

<class_name>Common::set_<prop_name>

Parameters

in <*prop_type*> new_value : the new value of the property. Typed to abstract interface **(<*prop_name*>Common**), if property is of class-type.

Semantics

If the property's type is primitive, the value is updated. If it is of class-type, merging is done according to Section 7.7, "Merging of repository instances," on page 63. Corresponds to an attribute's s*et*-operation in MOF, v1.4.

Returns

(none)

Exception

If an error occurs on update, **MofError** is raised. This may result due to a constraint violation or in terms of merging.

If a property's multiplicity lower = 0 and upper = 1, then the following unset-operation is derived:

<class_name>Common::unset_<prop_name>

Parameters

(none)

Semantics

Unsets the property. Corresponds to an attribute's *unset*-operation in MOF1.4.

Returns

(none)

Exception

MofError is raised if the element is not available.

Collections are mapped to abstract iterator interfaces as described in the following subsections.

Mapping to abstract interfaces

If a type (a data type or a classifier) appears as a property or an operation parameter with the upper value of the multiplicity being greater than 1, for this type is an abstract interface created. If the collection kind is Bag/Set/List/UList, this interface is defined by:

```
abstract interface
        <type of collection><collection kind>Iterator : MOF::IteratorBase {

        <type of collection> get_value ()
            raises (MofError);
        void insert_value ( in <type of collection> value )
            raises (MofError);
        void modify_value ( in <type of collection> value )
            raises (MofError);
        <type of collection><collection kind>Iterator next_one()
            raises (MofError);
        <type of collection><collection kind>Iterator previous_one()
            raises (MofError);
        <type of collection><collection kind>Iterator begin()
            raises (MofError);
        <type of collection><collection kind>Iterator end()
            raises (MofError);
        boolean is_empty() raises (MofError);
};
```

<type of collection><collection kind>Iterator::get_value

Parameters

(none)

Semantics

Returns the value of the element in the collection pointed to by this iterator. If this iterator is equivalent with the result of ***<type of collection><collection kind>*****Iterator::end()**, a MofError exception is raised.

Returns

Value of the element in the collection pointed to by this iterator.

Exception

MofError is raised if the iterator points to the last element in the list.

<type of collection><collection kind>Iterator::insert_value

Parameters

<type of collection> value

Semantics

Inserts the value in the parameter **value** into the collection. If the collection is ordered, the value is being inserted *before* the element this iterator points to, e.g., a call to **next_one** after a call to insert shall return the element the iterator points to before the **insert_value** operation. If the iterator points to the result of ***<type of collection><collection kind>*****Iterator::end()**, the value is being added at the end of the collection. If the iterator points to ***<type of collection><collection kind>*****Iterator::begin()**, the value is being inserted at the beginning of the collection.

Returns

(none)

Exception

MofError is raised if the collection this iterator points to is read-only or if an insertion of a value will increase the size of the collection to a value greater than upper bound – lower bound of the collection. MofError is also being raised, if the collection is unique and the element that shall be inserted is already existing in the collection.

<type of collection><collection kind>Iterator::modify_value

Parameters

<type of collection> value

Semantics

Sets the value of the collection to the one given in parameter value.

Returns

(none)

Exception

MofError is raised if the collection this iterator points to is read-only or the iterator points to the end of the collection. **MofError** is also being raised, if the collection is unique and the element that shall be inserted is already existing in the collection.

<type of collection><collection kind>Iterator::next_one

Parameters

(none)

Semantics

Returns an iterator to the element following the element pointed to by this iterator. If the element pointed to by this iterator is the last element in a collection, an iterator is being returned equivalent to the result of ***<type of collection><collection kind>*Iterator::end()**. The same holds if the operation is called at an iterator equivalent to the result of ***<type of collection><collection kind>*Iterator::end()**.

Returns

***<type of collection><collection kind>*Iterator**

Exception

(none)

<type of collection><collection kind>Iterator::previous_one

Parameters

(none)

Semantics

Returns an iterator to the element preceding the element pointed to by this iterator. If the element pointed to by this iterator is the first element in a collection, an iterator is being returned equivalent to the result of ***<type of collection><collection kind>*Iterator::begin()**. The same holds if the operation is called at an iterator equivalent to the result of ***<type of collection><collection kind>*Iterator::begin()**.

Returns

***<type of collection><collection kind>*Iterator**

Exception

(none)

<type of collection><collection kind>Iterator::begin

Parameters

(none)

Semantics

Returns an iterator pointing to the begin of the collection. If the one is equivalent with the result of ***<type of collection><collection kind>*****Iterator::end()**, the collection is empty.

Returns

<type of collection><collection kind>**Iterator**

Exception

(none)

<type of collection><collection kind>Iterator::end

Parameters

(none)

Semantics

Returns an iterator pointing to the end of the collection. If the one is equivalent with the result of ***<type of collection><collection kind>*****Iterator::begin()**, the collection is empty.

Returns

<type of collection><collection kind>**Iterator**

Exception

(none)

<type of collection><collection kind>Iterator::is_empty

Parameters

(none)

Semantics

Returns true if the collection the iterator points to is empty, false otherwise.

Returns

boolean

Exception

(none)

Mapping to concrete valuetypes and interfaces

Beside the derivation of an abstract interface for collections, a concrete valuetype is being created to enable the access by value semantics for collections. This valuetype implements all operations specified by the abstract interface it supports and provides a factory operation to construct a value based on a specific collection given as parameter to the factory operation.

```
valuetype <type of collection><collection kind>AsValue
        supports <type of collection><collection kind>Iterator {

        private <type of collection><collection kind> value;
        factory create ( in <type of collection><collection kind> initializer );
};
```

To allow access to the elements of a collection by reference, an interface is being produced that inherits from the abstract interface created for a collection. This interface inherits all operations defined by the abstract interface and extends them by an operation **as_value**. This operation returns the iterator as valuetype, pointing to the same element in the collection.

```
interface <type of collection><collection kind>AsReference
        : <type of collection><collection kind>Iterator
{
        <type of collection><collection kind>AsValue as_value()
                raises (MofError);
};
```

7.5.4 EMOF::Operation and EMOF::Parameter

An instance of the Operation class is mapped into the abstract interface that is derived for its owning class (see also CMOF::Operation for exceptional cases to the following):

<class_name>Common::<operation_name>

Parameters

The operation receives:

- An **in** parameter for each specified instance in p*arameter.*
- An **out** parameter for each specified instance of *returnResult* except for the first one.

Semantics

The computational semantic of the derived operation is user-specific. At least, the pre- and post-conditions must hold as specified in *precondition* and *postcondition*. Note that all appearing types are those of abstract interfaces derived for the origin classes.

Returns

Type of the abstract interface derived for the first instance in *returnResult* (or appropriate primitive type), **void** otherwise.

Exception

MofError is included in the raise clause. If the Operation specifies *raisedExceptions*, the elements thrown are included using the **element_in_error** member of the **MofError** exception.

7.5.5 EMOF::Tag

Tags are used to define service support. See Section 6.3.5, “Mapping of Tags and the MOF Services,” on page 30 for a discussion.

7.5.6 EMOF::Extent and EMOF::URIExtent

Extents and URIExtents are not mapped directly to IDL constructs. Instead, extents are rather implicitly derived from the package structure of a metamodel as in MOF1.4. See also Section 7.1, “Computational Semantics,” on page 41.

7.5.7 EMOF::MultiplicityElement

EMOF::MultiplicityElement is an abstract class, but the multiplcity of elements has influence on the general mapping. Details are defined in Section 7.5.3, “EMOF::Property,” on page 52.

NOTE: It should be clear, that every occurrence of multiplicities in the mapping leads to the iterator interfaces specified in 7.5.3.

7.5.8 EMOF::Enumeration and EMOF::EnumerationLiteral

Enumerations maps to IDL enum definitions and EnumerationLiterals to identifiers within the enumeration. All details are already given in Section 6.2.9, “Mapping of DataTypes and Enumerations,” on page 19.

7.6 Mapping for CMOF Compliant Models

The following subsections describe additions and differences to the EMOF mapping. For most elements, only small changes are needed.

7.6.1 CMOF::Package

The mapping for Packages is the same as for EMOF::Package.

7.6.2 CMOF::Class

The mapping for Class is the same as for EMOF::Class.

7.6.3 CMOF::Association

The mapping for Associations is optional. If it is mapped, an abstract interface, a component, a home, and a valuetype are derived. Inheritance maps to inheritance of the abstract interface. The operations in the abstract interface correspond to those of the MOF, v1.4 association interfaces:

<association_name>Common::all_<association_name>_links

Parameters

(none)

Semantics

Access to the complete link set via the returned iterator. Corresponds to *all_links*-operation of MOF, v1.4. If an association end is a union, all sub-association links are included in the return iterator.

Returns

LinkSetIterator to iterate over all links of the association.

Exception

MofError is raised if an error occurs.

<association_name>Common::create_link_in_<association_name>

Parameters

in <*association_name*>Link new_link: link to be inserted.

Semantics

A direct way to insert a link into the association's link-set. If the ends' multiplicity is one of [0..1]-[0..1], [0..1]-[1..1] or [1..1]-[1..1], the create_link operation provides the only way to add a new link. In the other cases, the link will be logically inserted at the end of the link-set (if ordering exists at one of the ends). Another way of inserting links is to use the iterator returned by the **linked_objects_*** operations. Comparable to *add*-operation of MOF, v1.4 (with modified execution semantic).

Returns

(none)

Exception

If the link *new_link* passed as parameter does already exists in the link set the operation raises a MofError with **DUPLICATE_VIOLATION** indicating the error. If the association specifies an end as derived union (isDerivedUnion = true), the create_link operation raises a **MofError** exception with **ADD_LINK_TO_UNION_VIOLATION**.

<association_name>Common::link_exists

Parameters

in <*association_name*>Link: link to check for

Semantics

Checks for whether a link is in the link-set of the association or not.

Returns

true, if the link is in the link set or, in case of union-ends, if it is in one of the subsets; false otherwise.

Exception

MofError is raised if an element or the link-set itself are unavailable.

<association_name>Common::remove_link

Parameters

in <*association_name*>Link link: link to be removed from the link-set.

Semantics

Operation removes **link** from the link-set, if it is in the set of links.

Returns

(none)

Exception

If link does not exists, the operation raises a **NotFound** exception. In case of a union end, the operation raises the **MofError** exception with **REMOVE_LINK_FROM_UNION_VIOLATION**

<association_name>Common::linked_objects_<end_name>

Parameters

in <*end_type*> <*end_name*> : instance of *end_type* to filter the link-set.

Semantics

Filters the link-set along the parameter instance and returns an iterator to all linked elements of this instance. The **linked_objects** operation replaces the former *add_**-operations of MOF1.4 with the comfortable iterator interface.

Returns

<*end_type*>{Bag|Set|List|UList}Iterator to access the links.

Exception

MofError is raised elements or links are not accessible.

7.6.4 CMOF::Property

In CMOF, mapping EMOF::Property is enhanced with respect to redefinitions of elements (as described in Section 6.2.12, “Mapping for Redefinitions,” on page 22). Details may be found in Section 7.6.16, “CMOF::Redefinition,” on page 62.

7.6.5 CMOF::Operation and CMOF::Parameter

Additionally to the mapping for EMOF::Operation, the CMOF concept of redefinition has strong influence on the typing an Operations parameters and return value.

The specification including an example is already detailed in section 6.2.12, ’Mapping for Redefinitions’.

7.6.6 CMOF::Exception

The mapping for Exceptions is discussed in Section 6.2.10, “Mapping of Exceptions,” on page 20 and in EMOF::Operation (Section 7.5.4, “EMOF::Operation and EMOF::Parameter,” on page 57).

7.6.7 CMOF::Tag

The mapping for Tags is the same as for EMOF::Tag.

7.6.8 CMOF::Extent and CMOF::URIExtent

Extents and URIExtents are not mapped to IDL (see also Section 7.5.6, “EMOF::Extent and EMOF::URIExtent,” on page 58).

7.6.9 CMOF::DataType, Enumeration, EnumerationLiteral

DataType instances are mapped to abstract interfaces and valuetypes in the same conceptual line as Classes. The mutator operations for properties and owned operations are identically mapped as in the case of a class. Note that MOF1.4 has no equivalent model element, since *StructureTypes* could not inherit nor support operations.

<data_type_name>::is_equal

Parameters

in <*data_type*> other_object: typed to derived valuetype and serves as object for comparison.

Semantics

Provides a deep-compare (recursive for all members) for value-equivalence.

Returns

true, if this data type has the same values as the passed one, false otherwise.

Exception

MofError is raised if compare fails or parameter is a nil reference.

<data_type_name>::creates

Parameters

Receives a parameter for each member (incl. base-types recursively).

Semantics

Creates an instance of the data type.

Returns

New data type instance.

Exception

MofError is raised if creation fails.

7.6.10 CMOF::Comment

Comments map as described in Section 6.2.15, “Mapping of other constructs,” on page 24.

7.6.11 CMOF::Constraint

Specifying constraints for a metamodel has no influence on the IDL mapping. Nevertheless, compliance points shall be provided as discussed in Section 6.2.14, “Mapping of Constraints,” on page 23.

7.6.12 CMOF::ElementImport

The mapping for ElementImport corresponds to the mapping of AliasTypes in MOF, v1.4.

For an ElementImport, an IDL **typedef** is declared in the module that owns the import:

typedef *<imported_element_qualified_name> <alias>*;

7.6.13 CMOF::OpaqueExpression and CMOF::Expression

Expressions do not map to CORBA IDL (cp. 6.2.15).

7.6.14 CMOF::PackageImport

The mapping for PackageImport implies aliasing of all elements in the imported package. *For each* element, an IDL **typedef** is declared in the module that corresponds to the importing package:

typedef *<imported_element_qualified_name> <element_name>*;

7.6.15 CMOF::PackageMerge

Package merge is regarded as a model transformation and hence only the (merged) result is mapped to IDL.

7.6.16 CMOF::Redefinition

A detailed discussion is already presented in Section 6.2.12, “Mapping for Redefinitions,” on page 22.

7.7 Merging of repository instances

Merging occurs in cases when previously created valuetypes are propagated back into the repository. A client is free to query an objects state, modify the received valuetype using supported operations locally and then send it back to the repository (e.g., update a CCM components state attribute).

The merging is based on the MofId (private) member of the valuetypes. The following rules apply:

1. The update of repository instances is *atomic* and succeeds completely or not at all (in the last case an exception is raised).
2. If a *copy_from* creation operation is invoked (e.g., in a components home) and a valuetype is passed that contains a MofId of an existing instance, then this instance is updated instead of creating a new instance.
3. If a state attribute is updated with a valuetype that has the same MofId as the object on which the update has been invoked, all owned properties are updated recursively that are contained by the passed valuetype (only contained valuetypes).
4. If within the recursive update process of 3. an object is no longer in the repository, it will be (re-)created (this is the same semantic as 2.) and obtains its old MofId.

8 Reflection Service

MOF 2.0 itself defines a reflective framework that serves as a basis for the reflective API of the IDL domain (cp. [8]). The given model elements are merged with interfaces according to MOF, v1.4 concepts to specify new IDL interfaces which the reflection service provides.

The integration of the interfaces is achieved using facets for the CCM mapping and inheritance in the Base-IDL mapping (details see Section 6.3.6, “Support for the Reflection Service,” on page 30 and Section 6.4.6, “Support of MOF Services and Reflection,” on page 36, respectively).

8.1 Base-IDL Reflection

As in the rest of the document, the reflective framework is designed on the one hand for EMOF and in an extended version for CMOF.

8.1.1 EMOF Reflective

The following figure shows the EMOF reflective interfaces. There exist specialized forms for packages, object (e.g., classes) and factories.

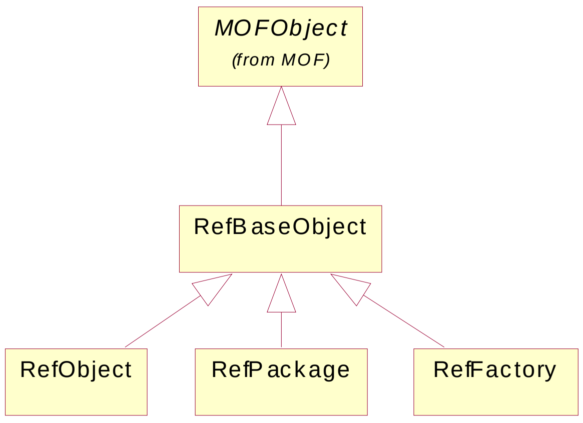


Figure 8.1- Inheritance within the EMOF reflective framework

For example, all interfaces that describe MOF-classes are inherited from RefObject (if it was requested at transformation time). The RefFactory interface is the supertype for all factories (Home interfaces or package factory interfaces). A factory is connected to one package and provides a collection of operations to create all possible types of the linked package.

Note that in principle we reuse the MOF, v2.0 Core semantics for reflection and only diverge from this to increase clearness (e.g., name clashing between CORBA::Object and CMOFReflection::Object.

RefBaseObject

This interface provides operations for all reflective interfaces. It is inherited by all other reflective interfaces.

RefBaseObject::ref_get_metaclass

Parameters

(none)

Semantics

An object is returned, that holds the metamodel specifications about the instance.

Returns

RefBaseObject : which describes this instance, a nil reference if it does not exist.

Exception

(none)

RefBaseObject::ref_container

Parameters

(none)

Semantics

The parent object, which contains this instance, is returned. If the instance is equal to the outermost package, a nil reference will be returned.

Returns

RefBaseObject : returns the parent object or nil reference, if the instance is not nested.

Exception

(none)

RefBaseObject::is_instance_of_type

Parameters

RefBaseObject type : type, which the instance should be tested with.

boolean include_subtypes : true, if all subtypes of the supplied type should be included in checking.

Semantics

Checks, if the type of the instance is equal to the supplied type. If include_subtypes is set to true, the test is recursively done over all subtypes of the supplied type.

Returns

boolean : true, if type matches (including subtypes, if flag is set to true). Otherwise false.

Exception

(none)

RefObject

The RefObject interface defines 4 operations for reflective access to properties, which are in EMOF always contained within a class and in CMOF also in associations, datatypes, primitive types, and enumerations. There are 3 different cases for modification in term of multiplicity:

- Single-valued property with default value
- Single-valued property without default value
- Multi-valued property

The IDL mapping for properties just generates attributes and operations to the specific IDL interfaces. So no explicit types for properties are available in the IDL domain. For unique classification of properties its name (as string) is used.

Thus it is unique within the scope of the property's element (e.g., a class).

In IDL, this is expressed using the following typedef:

typedef wstring PropertyName;

NOTE: During runtime the type of an instance is always determined at its creation time by the implementation of the factory. Thus, the PropertyName has a one-to-one relationship to a property, even if it is redefined.

RefObject::ref_get

Parameters

PropertyName prop : name of the property to request.

Semantics

When the property's multiplicity has an upper bound = 1, then the value of the property is returned. When the upper bound is > 1 an iterator (see Rule (3)) to a collection is returned, which is typed with the type of the property.

The returned collection is a

- Bag, if the prop is neither unique nor ordered.
- Set, if the prop is unique but not ordered.
- List, if the prop is ordered but not unique.
- UList, if the prop is ordered and unique.

Returns

any: the value of the property. Has to be narrowed to the actual type of the property.

Exception

If the property has no default value, and it has never been set (or the ref_unset operation was called before) a NotSet exception is raised.

A MofError with the error_code ILLEGAL_ARGUMENT_VIOLATION is thrown, if there is no property within the element, which has the same name as the supplied one.

RefObject::ref_set

Parameters

- ***PropertyName prop*** : the property to modify.
- ***any value*** : the new value for the property.

Semantics

This operation sets the new value of the property. If the property's multiplicity upper bound = 1, the value is directly set. When the property's upper bound is > 1 an iterator is expected, which points to a collection, which is of the property's type. The collection will overwrite the old one completely.

The supplied collection must be at least a

- UList, if the property is ordered and unique.
- Set, if the property is unique but not ordered.
- List, if the property is ordered but not unique.
- Bag, if the property is neither unique nor ordered.

NOTE: In case of passing a List containing duplicates for a property requiring a UList, the server is unable to decide which duplicated element to remove, because of the ordering. Therefore, this is up to the client.

But it can also be one of the collections, those are more restricted than the requested one. That means if a Bag must be provided, it is also possible to pass a List, Set, and UList. The implementation will guarantee that the collection will fit to the right type for later changes of the property.

Returns

(none)

Exception

A MofError with the error_code ILLEGAL_ARGUMENT_VIOLATION is thrown, if there is no property within the element, which has the same name as the supplied one.

If the property is set to be read only, a MofError with error_code NOT_CHANGEABLE_VIOLATION is thrown.

RefObject::ref_is_set

Parameters

- ***PropertyName prop*** : the property to check.

Semantics

If the property has multiplicity upper bound = 1, true is returned if the value of the property has been set after default initialization. That means, if the property is set to its default value (primitive types) or to nil (all reference types), the operation returns false. If the property has multiplicity upper bound >1, it returns true if the list is not empty.

Returns

boolean : true if value has been set before. Otherwise false.

Exception

A MofError with the error_code ILLEGAL_ARGUMENT_VIOLATION is thrown if there is no property within the element, which has the same name as the supplied one.

NOTE: Another interpretation of this operation could be that the property is only analyzed if it has the multiplicity [0..1]. In that case a MofError would be thrown if the multiplicity is different to [0..1]. The result would be true, if and only if the value is set. Otherwise false.

RefObject::ref_unset

Parameters

- ***PropertyName prop*** : the property to unset.

Semantics

Recovers the initial state (value at creation time) of the property. That means, if a default value for a primitive type has been defined in the metamodel, the property is set with the default value, else the property is set to a nil reference. If the property's upper bound is >1, the collection is cleared. If ref_is_set() is invoked directly after this operation, it will return false in all cases.

Already unset properties are not modified.

Returns

(none)

Exception

A MofError with the error_code ILLEGAL_ARGUMENT_VIOLATION is thrown if there is no property within the element, which has the same name as the supplied one.

RefPackage

In MOF, v1.4 this interface acts as a factory, which functionality has been moved to the RefFactory interface.

RefPackage::ref_factory

Parameters

(none)

Semantics

The IDL mapping adds an extra constraint to the relation between package and factory, so that there is only one factory per package.

A factory is always associated with one package.

In the CMOF2IDL mapping this operation returns a CMOF::Reflective::RefFactory as EMOF::Reflective::RefFactory, which the programmer has to narrow afterwards.

Returns

RefFactory : the factory of the package.

Exception

(none)

RefFactory

A RefFactory provides various methods for creation of all types, which are available in the linked package.

RefFactory::ref_package

Parameters

(none)

Semantics

The operation provides access to the underlying package, which defines the scope for those types, which could be created with the reflective operations of the linked RefFactory.

Returns

RefPackage : the factory for the package, or a nil reference if no factory is provided for this package.

Exception

(none)

RefFactory::ref_create_from_string

Parameters

- ***wstring from*** : name of the class to create an instance of. The format of the string is according to the ref_convert_to_string operation.

Semantics

The properties of the new instance, whose type is a primitive type, are set to their default values. All references to class types are set to a nil reference. If the property's upper bound is > 1 an iterator to an empty collection is set. The iterator and the collection will have the property's type and will conform to the ordered and unique characteristics of the property.

Returns

RefBaseObject : the new instance.

Exception

A MofError with error_code ILLEGAL_ARGUMENT_VIOLATION is raised, if there is no such object (defined with the supplied string) contained in the linked package.

RefFactory::ref_convert_to_string

Parameters

- ***RefBaseObject element*** : the object to calculate the name of.

Semantics

Defines a string to identify the object unique in the context of its parent container. The string is the name of the object.

Returns

wstring : the unique name of the object within its container.

Exception

A MofError with error_code ILLEGAL_ARGUMENT_VIOLATION is raised, if the linked package does not contain such an element.

RefFactory::ref_create

Parameters

- ***RefBaseObject class*** : class to build the new instance from.

Semantics

Creates an instance of the supplied object.

This operation has the same behavior in setting the properties of the new instance as the RefFactory::ref_create_from_string operation.

Returns

RefBaseObject : the new instance.

Exception

A MofError with error_code **ILLEGAL_ARGUMENT_VIOLATION** is raised, if the linked package does not contain such a class, to build the instance of.

8.2 CMOF Reflective

Because of the <<merge>> dependency between the EMOF and CMOF packages the reflective interfaces for CMOF will inherit from their corresponding interfaces in the EMOF Reflective package.

This does not conflict with the two <<combine>> dependencies from Reflection over CMOFReflection to CMOF, because through the inheritance the merge characteristic is preserved and all elements are available within CMOF, what is demanded in sense of a <<combine>>.

The figure below shows the inheritance between the CMOF and EMOF reflective framework.

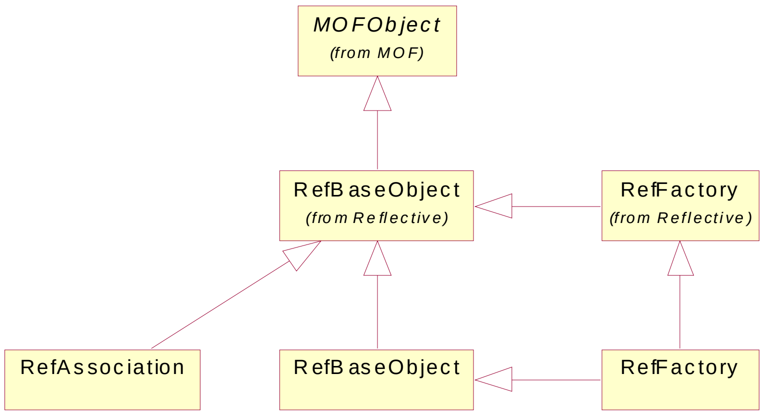


Figure 8.2 - Inheritance between the CMOF and EMOF reflective framework

In the CMOF reflective package a struct for name/value pairs is defined called RefArgument. This is necessary to provide reflective creation of objects with automatic initialization of their properties or for reflective invocations of operations.

```
struct RefArgument {
    wstring name;
    any value;
};

typedef sequence < RefArgument > ArgumentSet;
```

The above list has the advantage, that the order of the arguments is not relevant, because the arguments are identified through their name of RefArgument and supply their value within the value of RefArgument.

As described in Section 6.2.4, "Mapping of Collections (MultiplicityElement)," on page 14, it is recommended to use the iterator interfaces for the Reflective framework as well.

So the following iterators are defined for reflective usage:

```
interface ArgumentSetIterator : IteratorBase;

typedef sequence < RefBaseObject > RefBaseObjectSet;
interface RefBaseObjectSetIterator : IteratorBase;

typedef sequence < RefLink > LinkSet;
interface LinkSetIterator : IteratorBase;
```

The ArgumentSetIterator provides a more flexible control over passing and retrieving arguments, since the argument order is irrelevant (assignment of parameters is resolved via argument names).

The last collection is build of RefLink – a struct for representing instances of associations.

```
struct RefLink {
    RefBaseObject firstObject;
    RefBaseObject secondObject;
};
```

RefBaseObject

The CMOF::Reflective::RefBaseObject is a specialization of EMOF::Reflective::RefBaseObject.

In the IDL mapping there is no explicit type for operation, so they are unique identified with their names as string.

In IDL this is expressed with the following typedef:

```
    typedef wstring OperationName;
```

RefBaseObject::ref_invoke

Parameters

- ***OperationName operation*** **:** name of the operation to invoke.
- ***ArgumentSetIterator arguments*** **:** iterator of a name/value pairs list, which represents the arguments of the operation through the name of the RefArguments and their value.

Semantics

Calls the method specified by its name. The arguments of the operation are supplied via an iterator of an **ArgumentSet**. This list includes parameters of the operation identified with their name and providing their value within **RefArgument**. If an argument of the operation is not passed within the list, the default value is taken – if not present a nil reference is supplied instead.

If the invoked operation raises any exception, it will also be listed in the returned collection, but not raised.

Returns

RefBaseObjectSetIterator **:** iterator of a **RefBaseObjectSet**, which contains all return values of the invoked method.

Exception

A **MofError** with error_code **ILLEGAL_ARGUMENT_VIOLATION** is raised if the argument list contains at least one argument that does not fit to the arguments of the method to invoke. The same error is thrown, when the element contains no operation with the supplied name.

RefAssociation

All associations are inherited of this interface. It provides operation to create, alter, and query existing association instances.

RefAssociation::ref_all_links

Parameters

- ***boolean include_subtypes*** **:** controls whether to perform search for instances over all subtypes of the association.

Semantics

Collects all existing instances of the type of association the operation was called from. In MOF 2.0 associations can inherit from each other, so the operation provides the opportunity to perform the search including all subtypes of the association. The semantic is the same as described for the corresponding operation in Section 7.6.3, “CMOF::Association,” on page 59.

Returns

LinkSetIterator result **:** iterator to the set of RefLink, which will point at the resulting collection.

Exception

(none)

RefAssociation::ref_create_link

Parameters

- ***RefLink new_link*** **:** struct, that contains the first and second end for the new instance of the association.

Semantics

Constructs a link between the two supplied objects of the RefLink struct. The direction of the link is intuitive set through declaring the ends as either the first or the second end. The semantic is the same as described for the corresponding operation in Section 7.6.3, “CMOF::Association,” on page 59.

Returns

(none)

Exception

A **MofError** with error_code **ILLEGAL_ARGUMENT_VIOLATION** is raised, when one of the supplied objects within RefLink does not fit to the type of the association’s ends.

RefAssociation::ref_link_exists

Parameters

- ***RefLink link*** **:** struct, that contains the first and second end for the link to search for.

Semantics

Checks all instances of the association, and returns true, if an instance exists, which has the equal association ends in the right order as the supplied objects. The semantic is the same as described for the corresponding operation in Section 7.6.3, “CMOF::Association,” on page 59.

Returns

boolean **:** true, if link exists. Otherwise false.

Exception

(none)

RefAssociation::ref_remove_link

Parameters

- ***RefLink old_link* :** the link to delete.

Semantics

Deletes the supplied RefLink instance of the association. Note that the associated object instances are not deleted. The semantic is the same as described for the corresponding operation in Section 7.6.3, “CMOF::Association,” on page 59.

Returns

boolean : true, if link was successfully removed. Otherwise false.

Exception

Raises a NotFound exception, if the supplied link does not exist before the operation was called.

RefFactory

The CMOF RefFactory inherits from CMOF::Reflective::RefBaseObject and EMOF::Reflective::RefFactory.

It provides an enhanced operation for object creation.

RefFactory::ref_create_object

Parameters

- ***RefBaseObject class* :** determines the type of object that is being created.
- ***ArgumentSetIterator arguments* :** iterator of a name/value pairs list, which represents the properties of the object, identified by the name of the RefArguments that also supply their initial values.

Semantics

Creates an instance of the supplied object like the operation **EMOF::Reflective::RefFactory::ref_create**. But with the difference, that it is possible to supply initial values for the properties of the instance.

If there is no argument supplied for one of the objects properties, the property is set according to the operation **EMOF::Reflective::RefFactory::ref_create_from_string**.

Returns

***RefBaseObject* :** the new instance with the initialized properties.

Exception

Raises a MofError with error_code **ILLEGAL_ARGUMENT_VIOLATION**, if one of the supplied arguments is not a member of the object or the object to create the instance of is not contained in the linked package.

RefFactory::objects_of_type

Parameters

- ***RefBaseObject class* :** type of objects to search for.
- ***boolean include_subtypes* :** whether to perform a search over all subtypes of the supplied type, or not.

Semantics

This operation provides the opportunity to request all existing instances of a specified type. And if the flag include_subtype is set to true, all existing subtypes are included in the returned collection.

If the supplied type has the abstract flag set to true and include_subtypes = false, an empty collection will be returned.

Returns

RefBaseObjectSetIterator **:** iterator of a RefBaseObjectSet, which contains the requested objects.

Exception

Raises a **MofError** exception with error_code **ILLEGAL_ARGUMENT_VIOLATION**, if the supplied type is not contained within the linked package.

8.3 CCM Reflection

In the CCM framework built-in features are reused to handle some of the above described reflective tasks. So the resulting reflective API is thinned out essentially (e.g., the factory methods are provided in CCM through the respective home interfaces).

As in the Base-IDL reflection interfaces, the MOF2CCM mapping also defines basic reflective interfaces for EMOF and extends them for CMOF.

8.3.1 RefCCMBaseObject

The CCM reflective framework defines the RefCCMBaseObject as image of RefBaseObject in the EMOF Base-IDL mapping.

The three operations have been copied, but are typed to RefCCMBaseObject with the original semantic.

```
interface RefCCMBaseObject : MOFObject {
    RefCCMBaseObject get_meta_class ();
    RefCCMBaseObject get_container ();
    boolean is_instance_of_type ( in RefCCMBaseObject type,
                                  in boolean sub_types );
};
```

As in the Base-IDL mapping the CMOF reflective framework extends this interface with the opportunity for reflective invocation of methods.

```
RefCCMBaseObjectBagIterator ref_invoke ( in Operation operation, in ArgumentSetIterator
arguments );
```

For this method an iterator is defined as follows:

```
typedef sequence < RefCCMBaseObject > RefCCMBaseObjectSet;
interface RefCCMBaseObjectSetIterator : IteratorBase;
```

8.3.2 Home interfaces

The HomeFinder is a build-in CCM service that provides the opportunity to locate a component's CCMHome interface via the name of the component (see *CCMHome::find_home_by_name*).

Normally the programmer narrows the returned *CCMHome interface* downwards to the specific home to create a new component instance.

If the reflection service is supported, it is possible to narrow from *CCMHome* to *RefCCMHome*, which itself provides some common-typed operations to create component instances. The underlying implementation shall delegate the *RefCCMHome::create* operation to the specific *CCMHome* interface. But the return value of the *RefCCMHome::create* operation is common so that a client must not have any knowledge of the explicit type.

In the EMOF mapping the *RefCCMHome* interface defines a create operation, which is semantically equivalent to one of the create operations of *EMOF::Reflective::RefFactory.* The parameter to specify the type of the component (as **string** or through a class) is not necessary in CCM, because the scope of the specific component is already selected through one of *HomeFinder::find_home_by_** operations (see also notes on extents in Section 7.1, “Computational Semantics,” on page 41).

RefComponent create ();

In CMOF this interface is enhanced with the create method that provides simultaneous initialization of the component's properties as defined in the *CMOF::Reflective::RefFactory.*

RefComponent create (in ArgumentSetIterator arguments);

Through the selected return type of *RefComponent* it is ensured that the reflective API conforms to the CCM domain by using homes to create instances of a component type.

8.3.3 RefCCMObject

As *RefObject* from the Base-IDL mapping the *RefCCMObject* interface provides operations to change a component's properties. The syntax and semantic is completely the same as their clones in *EMOF::Reflective::RefObject* interface of the Base-IDL mapping, but they are typed to *RefCCMBaseObject.*

```
interface RefCCMObject : RefCCMBaseObject {
    RefCCMBaseObject ref_get ( in Property property )
            raises (MofError, NotSet);
    void ref_set ( in Property property,
                    in RefCCMBaseObject object )
                        raises (MofError);
    boolean ref_is_set ( in Property property ) raises (MofError);
    void ref_un_set ( in Property property ) raises (MofError);
};
```

8.3.4 RefCCMAssociations

The RefCCMAssociation interface is an exact image of the *RefAssociation* of the Base IDL mapping.

8.4 Reflective IDL (complete)

```
module MOF {
module EMOF {
module Reflective {

    typedef wstring Property;

    interface RefBaseObject;
    interface RefFactory;

    interface RefBaseObject : MOFObject {
        RefBaseObject get_meta_class ();
        RefBaseObject get_container ();
        boolean is_instance_of_type ( in RefBaseObject type,
                                      in boolean sub_types );
    };

    interface RefObject : RefBaseObject {
        any ref_get ( in Property prop ) raises (MofError, NotSet);
        void ref_set ( in Property prop, in any new_value ) raises (MofError);
        boolean ref_is_set ( in Property prop ) raises (MofError);
        void ref_un_set ( in Property prop ) raises (MofError);
    };

    interface RefPackage : RefBaseObject {
        RefFactory ref_factory ();
    };

    interface RefFactory : RefBaseObject {
        RefPackage ref_package ();
        RefBaseObject ref_create_from_string ( in wstring from ) raises (MofError);
        wstring ref_convert_to_string ( in RefBaseObject element )
                raises (MofError);
        RefBaseObject ref_create ( in RefBaseObject class ) raises (MofError);
    };

}; // Reflective
module CCMReflective {

    typedef wstring Property;

    interface RefCCMBaseObject : MOFObject {
        RefCCMBaseObject get_meta_class ();
        RefCCMBaseObject get_container ();
        boolean is_instance_of_type ( in RefCCMBaseObject type,
                                      in boolean sub_types );
    };
    interface RefCCMObject : RefCCMBaseObject , Components::CCMObject {
        any ref_get ( in Property prop ) raises (MofError, NotSet);
        void ref_set ( in Property prop, in any new_value ) raises (MofError);
        boolean ref_is_set ( in Property prop ) raises (MofError);
        void ref_un_set ( in Property prop ) raises (MofError);
    };
    interface RefCCMHome : Components::CCMHome {
        RefCCMObject ref_create ();
    };
}; // CCMReflective
}; // EMOF

module CMOF {
module Reflective {

    typedef wstring Operation;
    interface RefBaseObject;

    struct RefArgument {
        wstring name;
        any element;
    };
    typedef sequence < RefArgument > ArgumentSet;
    interface ArgumentSetIterator : IteratorBase {
        RefArgument get_value () raises (MofError);
        void set_value ( in RefArgument value ) raises (MofError);
        ArgumentSetIterator next_one () raises (MofError);
        ArgumentSetIterator previous_one () raises (MofError);
        ArgumentSetIterator begin () raises (MofError);
        ArgumentSetIterator end () raises (MofError);
    };
    valuetype ArgumentSetAsValue supports ArgumentSetIterator {
        private ArgumentSet ArgumentSetvalue;
        factory create ( in ArgumentSet initializer );
    };
    interface ArgumentSetAsReference : ArgumentSetIterator {
        ArgumentSetAsValue as_value () raises (MofError);
    };

    struct RefLink {
        RefBaseObject firstObject;
        RefBaseObject secondObject;
    };
    typedef sequence < RefLink > LinkSet;
    interface LinkSetIterator : IteratorBase {
        RefLink get_value () raises (MofError);
        void set_value ( in RefLink value ) raises (MofError);
        LinkSetIterator next_one () raises (MofError);
        LinkSetIterator previous_one () raises (MofError);
        LinkSetIterator begin () raises (MofError);
        LinkSetIterator end () raises (MofError);
    };
    valuetype LinkSetAsValue supports LinkSetIterator {
        private LinkSet LinkSetvalue;
        factory create ( in LinkSet initializer );
    };
    interface LinkSetAsReference : LinkSetIterator {
        LinkSetAsValue as_value () raises (MofError);
    };

    typedef sequence < RefBaseObject > RefBaseObjectSet;
    interface RefBaseObjectSetIterator : IteratorBase {
        RefBaseObject get_value () raises (MofError);
        void set_value ( in RefBaseObject value ) raises (MofError);
        RefBaseObjectSetIterator next_one () raises (MofError);
        RefBaseObjectSetIterator previous_one () raises (MofError);
        RefBaseObjectSetIterator begin () raises (MofError);
        RefBaseObjectSetIterator end () raises (MofError);
    };
    valuetype RefBaseObjectSetAsValue supports RefBaseObjectSetIterator {
        private RefBaseObjectSet RefBaseObjectSetvalue;
        factory create ( in RefBaseObjectSet initializer );
    };
    interface RefBaseObjectSetAsReference : RefBaseObjectSetIterator {
        RefBaseObjectSetAsValue as_value () raises (MofError);
    };

    interface RefBaseObject : MOF::EMOF::Reflective::RefBaseObject {
        RefBaseObjectSetIterator ref_invoke ( in Operation op,
                        in ArgumentSetIterator arguments )
                            raises (MofError);
    };

    interface RefAssociation : RefBaseObject {
        LinkSetIterator ref_all_links ();
        void ref_create_link ( in RefLink new_link ) raises (MofError);
        boolean ref_link_exists ( in RefLink link ) raises (MofError);
        boolean ref_remove_link ( in RefLink old_link )
                        raises (MofError, NotFound);
        LinkSetIterator ref_linked_object ( in RefBaseObject end,
                        in boolean direction ) raises (MofError);
    };

    interface RefFactory : RefBaseObject, ::MOF::EMOF::Reflective::RefFactory {
        RefBaseObject ref_create_object ( in RefBaseObject class,
                        in ArgumentSetIterator arguments ) raises (MofError);
        RefBaseObjectSetIterator ref_objects_of_type ( in RefBaseObject class,
                        in boolean include_subtypes ) raises (MofError);
    };
```

```
}; // Reflective
module CCMReflective {
    typedef wstring Operation;
    interface RefCCMBaseObject;
    struct RefArgument {
        wstring name;
        any element;
    };
    typedef sequence < RefArgument > ArgumentSet;
    interface ArgumentSetIterator : IteratorBase {
        RefArgument get_value () raises (MofError);
        void set_value ( in RefArgument value ) raises (MofError);
        ArgumentSetIterator next_one () raises (MofError);
        ArgumentSetIterator previous_one () raises (MofError);
        ArgumentSetIterator begin () raises (MofError);
        ArgumentSetIterator end () raises (MofError);
    };
    valuetype ArgumentSetAsValue supports ArgumentSetIterator {
        private ArgumentSet ArgumentSetvalue;
        factory create ( in ArgumentSet initializer );
    };
    interface ArgumentSetAsReference : ArgumentSetIterator {
        ArgumentSetAsValue as_value () raises (MofError);
    };

    struct RefLink {
        RefCCMBaseObject firstObject;
        RefCCMBaseObject secondObject;
    };
    typedef sequence < RefLink > LinkSet;
    interface LinkSetIterator : IteratorBase {
        RefLink get_value () raises (MofError);
        void set_value ( in RefLink value ) raises (MofError);
        LinkSetIterator next_one () raises (MofError);
        LinkSetIterator previous_one () raises (MofError);
        LinkSetIterator begin () raises (MofError);
        LinkSetIterator end () raises (MofError);
    };
    valuetype LinkSetAsValue supports LinkSetIterator {
        private LinkSet LinkSetvalue;
        factory create ( in LinkSet initializer );
    };
    interface LinkSetAsReference : LinkSetIterator {
        LinkSetAsValue as_value () raises (MofError);
    };

    typedef sequence < RefCCMBaseObject > RefCCMBaseObjectSet;
    interface RefCCMBaseObjectSetIterator : IteratorBase {
        RefCCMBaseObject get_value () raises (MofError);
        void set_value ( in RefCCMBaseObject value ) raises (MofError);
        RefCCMBaseObjectSetIterator next_one () raises (MofError);
        RefCCMBaseObjectSetIterator previous_one () raises (MofError);
        RefCCMBaseObjectSetIterator begin () raises (MofError);
        RefCCMBaseObjectSetIterator end () raises (MofError);
    };
    valuetype RefCCMBaseObjectSetAsValue supports RefCCMBaseObjectSetIterator
{
        private RefCCMBaseObjectSet RefCCMBaseObjectSetvalue;
        factory create ( in RefCCMBaseObjectSet initializer );
    };
    interface RefCCMBaseObjectSetAsReference : RefCCMBaseObjectSetIterator {
        RefCCMBaseObjectSetAsValue as_value () raises (MofError);
    };

    interface RefCCMObject : MOF::EMOF::CCMReflective::RefCCMObject {
        RefCCMBaseObjectSetIterator ref_invoke ( in Operation op,
                in ArgumentSetIterator arguments ) raises (MofError);
    };
    interface RefCCMAssociation : RefCCMObject,
      MOF::EMOF::CCMReflective::RefCCMObject {
        LinkSetIterator ref_all_links ();
        void ref_create_link ( in RefLink new_link ) raises (MofError);
        boolean ref_link_exists ( in RefLink link ) raises (MofError);
        boolean ref_remove_link ( in RefLink old_link ) raises (NotFound);
        LinkSetIterator ref_linked_object ( in RefCCMBaseObject end,
                in boolean direction ) raises (MofError);
    };
    interface RefCCMHome : EMOF::CCMReflective::RefCCMHome {
                RefCCMObject ref_create_and_init ( in ArgumentSetIterator arguments )
                        raises (MofError);
    };

}; // CCMReflective
}; // CMOF
}; // MOF
```

Annex A
(normative)

References

[1] Meta Object Facility (MOF) Specification, Version 1.4, OMG document ptc/2001-10-04

[2] MOF 2.0 to OMG IDL Mapping RFP, OMG document ad/2001-11-07

[3] MOF 2.0 Facility and Object LifeCycle RFP, OMG document ad/03-01-35

[4] MOF 2.0 Query/View/Transformation RFP, OMG document ad/02-04-10

[5] MOF 2.0 Versioning and Development Lifecycle RFP, OMG document ad/06-23

[6] MOF 2.0 Core RFP, OMG document ad/2001-11-05

[7] CORBA 3.0: New Components Chapters, OMG TC Document ptc/2001-11-03

[8] MOF 2.0 Core revised submission, OMG document ad/03-04-07

[9] U2P UML Infrastructure, OMG document ad/03-03-01

[10] www.puml.org/mml

[11] MOF Query/View/Transformations Initial submission, OMG document ad/03-02-03

[12] XMOF, OMG document ad/03-03-24

[13] MOF2.0 to IDL as OCL constraints convienience document, OMG document ad/04-01-11

INDEX