MOF 2.0 IDL Specification

This OMG document replaces the submission document (ad/03-10-03) and the Draft Adopted specification (ptc/05-03-05). It is an OMG Final Adopted Specification and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by June 30, 2005.

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

The FTF Recommendation and Report for this specification will be published on November 18, 2005. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

OMG Adopted Specification
ptc/05-11-08

1 Scope

This specification defines the MOF 2.0 to CORBA IDL mapping. The mapping rules are based on the MOF2.0 Core document ptc/04-10-15 [6]. The set of mapping rules is outlined in section 6 and defined in detail in section 7.

This mapping reuses 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 submitters believe that this specification benefits from the gained experiences during several years in developing and using metadata repositories and CORBA technology.

2 Conformance

An implementation that conforms to the MOF 2 IDL specification must conform to one of the compliance points in [6] 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 6.3.2, "EMOF CCM Mapping Rules," on page -22.
- EMOF-to-Base-IDL: EMOF compliant models mapped using the Base-IDL mapping rules defined in Section 6.3.3, "EMOF Base-IDL Mapping," on page -29.
- CMOF-to-CCM: CMOF compliant models mapped using the CCM mapping rules defined in Section 6.4.2, "CMOF CCM Mapping Rules," on page -36.
- CMOF-to-Base-IDL: CMOF compliant models mapped using the Base-IDL rules defined in Section 6.4.3, "CMOF Base-IDL Mapping Rules," on page -38

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 are specific (Section 7.3, "Global Definitions for EMOF and CMOF," on page -43 "set_call_semantic" and "set_value_depth" operations).

3 Normative References

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

- Object Management Group, MOF 2.0 Core Adopted Specification, ptc/04-10-15.
- Object Management Group, UML 2.0 Infrastructure Final Adopted Specification, ptc/03-09-15.

4 Additional Information

4.1 Acknowledgements

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 submitters would like to thank the participants of this project for their contributions and review activities.

5 Methodology

This chapter describes the submitters' approach for the MOF 2.0 IDL language mapping. Due to the fact that the mapping is strongly dependent on the MOF 2.0 Core and other currently ongoing finalization processes (such as MOF2.0 QVT [4], MOF2.0 Versioning and Development Lifecycle [5], etc), recent results of these specifications could not be reflected in this mapping.

5.1 Overview

The key to the MOF2.0 IDL language mapping is 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 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.

5.2.1 Separation of MOF and Service Mapping

This specification separates the mapping of core MOF2.0 concepts from those of services. Although Reflection is available in MOF2.0 Core, it is considered in the mapping as a standard service. Non standard services can be plugged into the mapping through MOF Tags.

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.

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.

5.2.2 The MOF Model

The MOF Model as described in [6] is devided into EMOF and CMOF packages constructed by means of package merge. The resulting EMOF model is shown below for convenience. For CMOF, refer to [6]. A proposed mapping onto IDL elements is provided later in this specification in chapters 6 and 7.

NOTE: The EMOF class diagrams is included for convenience here. In case of any inconsistencies with the MOF2.0 Core Specification, see chapter 3, page 1 for the valid normative reference.

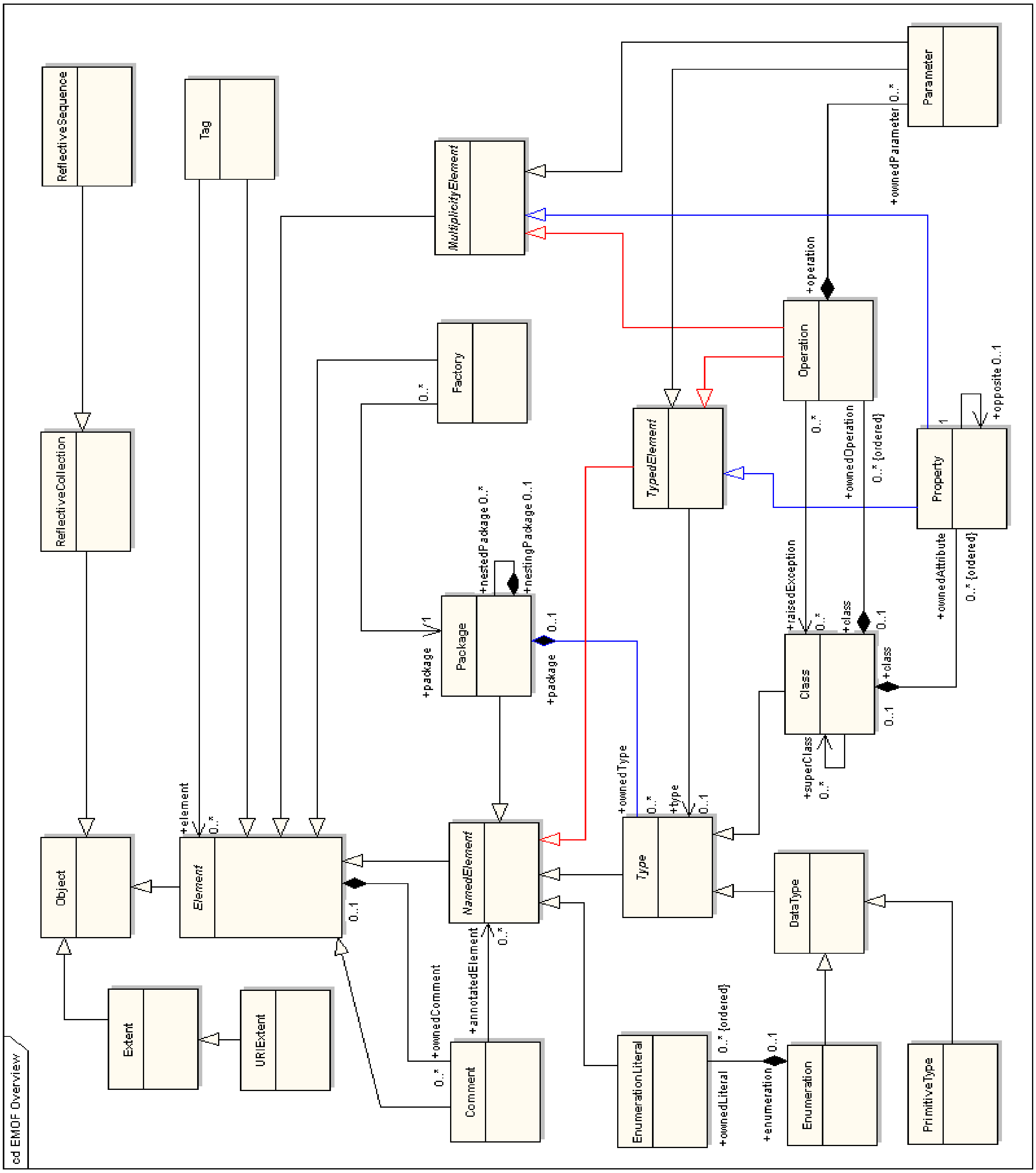


Figure 1 - The EMOF Model Overview

5.2.3 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 [6] (section 9) 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 6.3.2.10, “Support for the Reflection Service,” on page 27 and details in chapter 8.

5.3 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 5.2.3, “Reflection,” on page 6)
- Versioning and User Management
- Transactions
- LifeCycle
- ActiveRepository (Event based communication, see Section 6.3.2.11, “Support for Notification-Based Model Change Communication,” on page 28)
- 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 appropiate 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.

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 an improvement of the MOF1.4 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 [6] and 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 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 dropped 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 RFP.

6.2 General Mapping Rules

The general mapping rules apply to both EMOF and CMOF mappings. They concentrate on aspects that are specific to neither EMOF and CMOF such as formatting of identifiers, global interfaces and mapping of primitive types.

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]. In format 1, the first letter of each word is converted into upper case, and all other letters remain the same as input. It is used to produce names of modules and interfaces.
- **identifier format_2 (<MOF model element identifier>)** returns an identifier for an IDL definition according to 5.7.1.3 of [1]. In format 2, all letters in each word are converted to lower case and all words are separated by an underscore (“_”). It is used to produce names of attributes, operations and formal parameters and members of structured data types.
- **identifier format_3 (<MOF model element identifier>)** returns an identifier for an IDL definition according to 5.7.1.4 of [1]. In format 3, all letters in each word are converted to upper case and all words are separated by an underscore (“_”). It is used to produce names of constants.
- **identifier concatenate (string, string)** returns a concatenation of the two parameters.

Table 1 on page -13 shows some examples.

Table 1 - Identifier Formats

Name	Name split into words	Identifier in format 1	Identifier in format 2	Identifier in format 3
foo	“foo”	Foo	foo	FOO
foo_bar	“foo” “bar”	FooBar	foo_bar	FOO_BAR
ALPHAbetOrder	“ALPHAbet” “Order”	ALPHAbetOrder	alphabet_order	ALPHABET_ORDER
-a1B2c3_d4_	"a1" "B2c3" "d4"	A1B2c3D4	a1_b2c3_d4	A1_B2C3_D4
IKV pty ltd	"IKV" "pty" "ltd"	IKVtyLtd	dstc_pty_ltd	IKV_PTY_LTD

6.2.2 Globals

To provide a common base interface for every derived MOF model element, we introduce an (abstract) base interface equivalent to Object in MOF2.0 Core.

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 equals( 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 **equals** and **delete** operations are needed for identity support and destruction of objects, since abstract interfaces do not inherit from **CORBA::Object**[1].

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;
};
```

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

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

In MOF 1.4, 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 propose to use a generic **MOFIterator** defined in the MOF IDL module. The iterator interface declares the following 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.
- **is_at_end** to test whether the iterator is pointing to the last value in the collection.

Since the only common base type in IDL is CORBA::Any (which might not be very manageable for clients in querying collections), the type of the iterator interface is further distinguished between object types and primitive types. Furthermore, valuetypes are introduced to retrieve the collection in a convenient way in cases clients want to operate in pure local manner. For all objects (i.e. MOFObjects), the iterator interface is:

```
abstract interface MOFIterator : MOFObject
{
    boolean is_empty();
    boolean is_at_end();
};

abstract interface MOFObjectIterator : MOFIterator {
   MOFObject get_value () raises (MofError);
   void insert_value ( in MOFObject value ) raises (MofError);
   void modify_value ( in MOFObject value ) raises (MofError);
   MOFObject remove_value () raises (MofError);
   MOFObjectIterator next_one() raises (MofError);
   MOFObjectIterator previous_one() raises (MofError);
```

```
    MOFObjectIterator begin() raises (MofError);
    MOFObjectIterator end() raises (MofError);
};

typedef sequence < MOFObject > MOFObjectCollection;
valuetype MOFObjectIteratorAsValue : MOFState supports MOFObjectIterator {
    private MOFObjectCollection objects;
    factory create( in MOFObjectCollection initializer );
};

interface MOFObjectIteratorAsRef : MOFObjectIterator {
    MOFObjectIteratorAsValue as_value() raises (MOFError);
};
```

For the primitive types **Integer**, **String**, **Boolean** and **UnlimitedNatural** the following iterators are provided:

```
abstract interface <primitive_type>Iterator : MOFIterator {
    <primitive_type> get_value () raises (MofError);
    void insert_value ( in <primitive_type> value ) raises (MofError);
    void modify_value ( in <primitive_type> value ) raises (MofError);
    <primitive_type> remove_value () raises (MofError);
    <primitive_type>Iterator next_one() raises (MofError);
    <primitive_type>Iterator previous_one() raises (MofError);
    <primitive_type>Iterator begin() raises (MofError);
    <primitive_type>Iterator end() raises (MofError);
};

typedef sequence < <primitive_type_as_idl_type> > <primitive_type_name>Collection;
valuetype <primitive_type>IteratorAsValue : MOFState supports <primitive_type>Iterator {
    private <primitive_type>Collection values;
    factory create( in <primitive_type>Collection initializer );
};

interface <primitive_type>IteratorAsRef : <primitive_type>Iterator {
    <primitive_type>IteratorAsValue as_value() raises (MOFError);
};
```

The interfaces of the collections themselves, **ReflectiveCollection** and **ReflectiveSequence** provided by MOF2.0 Core are extended by the **get_iterator()** operation which returns a **MOFIterator**. Thereby, clients can simply get an iterator instance (by reference or by value) and iterate over the collection:

```
interface ReflectiveCollection
{
     MOFIterator get_iterator() raises (MofError);
     // other MOF2 core operations.
};
```

The returned **MOFIterator** reference is then of a more specific type **MOFObjectIterator** or primitive iterator type.

Because IDL does not support overloading, the “add” and “remove” operations in ReflectiveSequence are suffixed with “_at”.

```
interface ReflectiveSequence : ReflectiveCollection
{
    void add_at(in long index, in any object) raises (MofError);
    any remove_at(in long index) raises (MofError);
    // other MOF2 core operations.

};
```

Furthermore, the repository needs to provide the means of obtaining an empty reflective collection or sequence to use when creating new objects that have attributes with multiplicity upper bound > 1. This may be provided in an initial interface to the repository.

```
interface MofRepositoryRoot {
    ReflectiveCollection create_collection( in boolean is_ordered );

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

The is_ordered boolean parameter indicates whether a ReflectiveCollection (FALSE) or ReflectiveSequence (TRUE) should be created.

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 14.3 of [6] (CMOF Constraints) for the primitive types.

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 e.g. for an apropriate IDL **typedef** definition in the module to define the primitive type.

6.2.4 Mapping of Collections (MultiplicityElement)

All operations to get access to attributes with multiplicity upper > 1, will use either the ReflectiveCollection or ReflectiveSequence interfaces as return type (depending on orderedness). Internally the ReflectiveCollections or ReflectiveSequences hold a collection of objects (by reference or value).

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 and iterate over the collection.

NOTE: Although the word “Reflective” appears in the collection and iterator interface names, they are not used used only together with the reflection service.

An example is given in Section 6.3.2.3, “Mapping of Attributes and Operations,” on page 23.

6.3 EMOF Mapping Rules

6.3.1 EMOF Common Mapping Rules

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

6.3.1.1 Mapping of Packages

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 (2) 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 (3) 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.

6.3.1.2 Mapping of Classes

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 (4) A Class of a MOF 2.0 compliant model is mapped to a CORBA IDL abstract interface, named **format_1 (<class identifier>)**, 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).

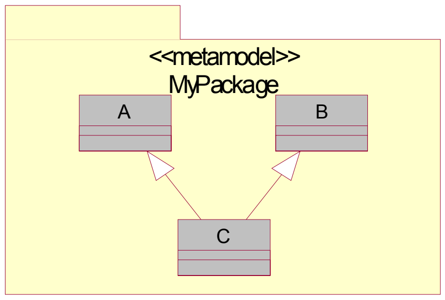
Rule (5) In the same module, a CORBA IDL valuetype is declared with the name **concatenate (format_1 (<class identifier>), "State")**, that supports the abstract interface generated following Rule (4) and inherits from **MOFState (truncatable).**

6.3.1.3 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 (6) If classes in a model inherit from another, this relation is mapped to inheritance relations between the respective abstract interfaces generated according to Rule (4). The valuetypes generated according to Rule (5) 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 A : MOFObject { };
    abstract interface B : MOFObject { };
    abstract interface C
      : A, B { };
    valuetype AState : truncatable MOFState
      supports A { };
    valuetype BState : truncatable MOFState
      supports B { };
    valuetype CState : truncatable MOFState
       supports C { };
} ;
```

6.3.1.4 Mapping of Attributes and Operations

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 6.4.1.5, "Mapping for Redefinitions," on page 34).

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).

Rule (7) The generated IDL operations for attributes/operations of a MOF model are

Rule (8) 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 (9) For an attribute with multiplicity upper bound = 1, two operations to set and get the attributes value are declared - taking into account Rule (7). 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 (5) 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 (10) Furthermore, the IDL valuetype generated following Rule (5) 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 (11) The definition of Rule (10) 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 proposed iterator mapping (see Section 6.2.4, "Mapping of Collections (MultiplicityElement)," on page 17) also applies for attributes with multiplicity upper > 1, we extend the mapping rule defined by Rule (9) with the following:

Rule (12) For an attribute with multiplicity upper>1, two operations to set and get the attribute value are declared - taking into account Rule (7).

The operation to get the value of the attribute has the name **concatenate ("get_", format_2 (<attribute identifier>))**. The return type is either ReflectiveCollection or ReflectiveSequence (depending on the orderedness of the property).The operation to set the value has the name **concatenate ("set_", format_2 (<attribute identifier>))**. The operation has one **in**-parameter of type ReflectiveCollection or ReflectiveSequence (depending on the orderedness of the property).

Rule (13) Furthermore, we extend the definition of Rule (10) for the case, that the multiplicity has an upper bound > 1, the IDL valuetype generated according to Rule (5) is extended by a private state member with the name **format_2 (<attribute identifier>)**. The type of the member is either ReflectiveCollection or ReflectiveSequence (depending on the orderedness of the property).

For specification details, see chapter 7. A comprehensive example especially for the multiplicity is given in Example (3).

Property modifications

This subsection defines alterations in the normal mapping (i.e., (Rule (7) to Rule (13)) with respect to attributes and associated elements of a property in MOF2. The specification details can be found in chapter 7.

Rule (14) 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 (25) and Rule (26) / Rule (52)). However, the attributes state may be changed by operations that exist for the instance.
- **{isID = true}** : if a class owns a property with the isID flag set to true, the computational semantic of the inherited

equals operation is altered (from **MOFObject**). In this case, identity is established via this property (the MOF2 abstract semantic is used for the mapping), i.e.:

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 it's of classifier type, then the delete operation also destroys these referenced instance(s).
- **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, no mapping is provided.
- ***{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. 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.

6.3.1.5 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 (15) 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 (2). 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 6.3.1.3, "Mapping of Inheritance between Classes," on page 18).

Rule (16) 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 (17) 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 (7) and Rule (9). Furthermore, the valuetype derived in Rule (15) always contains two operations:

- **boolean equals (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 (18) Since enumerations are constrained to have no operations or properties (cp. constraints in [6]), instances of class Enumeration are mapped to an IDL **enum** (defined in the module that is derived for the containing package according to Rule (2)) 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 *ownedLiteral* collection of the model.

Rule (19) 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 6.4.1.5, “Mapping for Redefinitions,” on page 34).

6.3.1.6 Mapping of Exceptions

The lack of an IDL exception not being able to inherit from other exceptions[1] leads to the decision, to reuse the exception framework of MOF1.4. With some minor improvements, the elementary exception remains still **MofError**:

```
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 everthing that is of type Class, the exception mapping is more complicated than in MOF1.4.

Rule (20) 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 apropriate instance.

6.3.1.7 Mapping of Reflection

Reflection is regarded as MOF Service (orthogonal to any meta-model) and thus mapped as described in Section 6.3.2.10, “Support for the Reflection Service,” on page 28 and Section 6.3.3.10, “Support for the Reflection Service,” on page 33.

6.3.1.8 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 MOF1.4 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 meta-models. This has already been done in MOF1.4.

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

- **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 MOF2.0 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 initalized 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.

In this mapping, a **verify** operation has been provided in the MOFObject interface, to which OCL expresssions can be passed (see Section 6.2.2, "Globals," on page 13).

6.3.1.9 Mapping of Comments

Rule (21) If a comment is attached to an element of type Class, Association (CMOF), 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.

6.3.1.10 Mapping of Extents and URIExtents

Extents and ***URIExtents*** are not mapped to IDL.

6.3.1.11 Mapping of Tags

Since tags are the MOF's extension mechanism, we will distinguish between the CCM and Base-IDL mapping. Refer to Section 6.3.2.9, "Mapping of Tags," on page 28 and Section 6.3.3.7, "Mapping of Tags," on page 33.

6.3.2 EMOF CCM Mapping Rules

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

6.3.2.1 Mapping of Classes

Apart from the abstract interfaces and valuetypes derived for a class according to Rule (4) and Rule (5), component type definitions are generated, representing the class's instances.

Rule (22) If the class is not abstract, an IDL component definition is being generated in the same module as the abstract interface (cp. Rule (4)) with the name **concatenate(format_1 (<class identifier>), “Component“)**. The component is declared to support the generated abstract interface for the class.

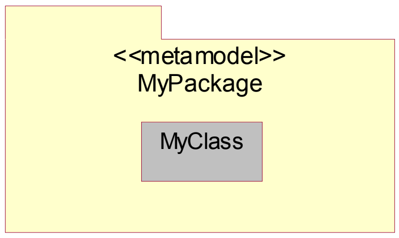
Rule (23) The IDL component definition generated according to Rule (22) is extended by an attribute with the name **concatenate (format_2 (<class identifier>), "_state")**. The type of this attribute is the IDL valuetype generated according to Rule (5).

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 59.

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 (24) 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 (22).

Example (2) For a class **MyClass** defined in the package **MyPackage**, the following CORBA IDL definition is being generated:



```
module MyPackage
{

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

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

6.3.2.2 Mapping of Inheritance of between Classes

Inheritance of classes is solely realized by the inheritance of the derived abstract interfaces (Section 6.3.1.3, “Mapping of Inheritance between Classes,” on page 18). A component’s support of its abstract interface assures type consistency.

6.3.2.3 Mapping of Attributes and Operations

While the mapping of attributes and operations for the common interface is already defined in Section 6.3.1.4, “Mapping of Attributes and Operations,” on page 18, only the creation of instances remains to be specified.

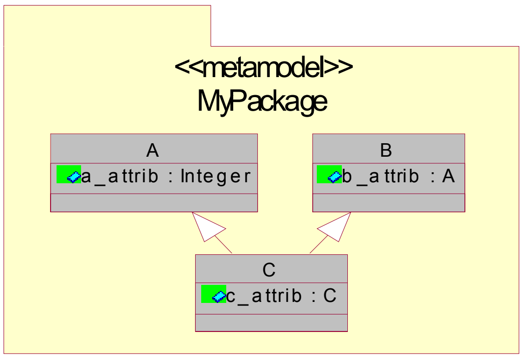
The home interface playing the role of a factory for a class’s instances in a repository is being extended with two more factory operations: One that initializes the created instance with values for all attributes from parameters of the factory operation.

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 (5) or call it with a component reference of the same or more specific type.

Rule (25) If a class in the model contains non-derived attributes, the home interface declaration generated according to Rule (24) 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 (26) 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 (2).

Example (3) Let **A**, **B** and **C** be classes in the package **MyPackage**. **C** may inherit from **A** and **B**. **A** may have an public attribute **a_attrib** with multiplicity [1..1] of type **Integer**. **B** may have an 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 A : MOFObject {
        long get_a_attrib ();
        void set_a_attrib ( in long a_attrib );
    };
    valuetype AState : truncatable MOFState
      supports A {
        private long a_attrib;
    };
    component AComponent supports A {
        attribute AState a_state;
    } ;
    home AHome manages AComponent {
        factory create_and_init_a ( in long a_attrib );
        factory copy_from ( in A the_a );
    };

    abstract interface B : MOFObject {
        A get_b_attrib ();
        void set_b_attrib ( in A b_attrib );
    };
    valuetype BState : truncatable MOFState
      supports B {
        private A b_attrib;
    };
    component BComponent supports B {
        attribute BState b_state;
    };
    home BHome manages BComponent {
        factory create_and_init_b
            ( in A b_attrib );
        factory copy_from ( in B the_b );
    };
    abstract interface C
      : A, B {
        C get_c_attrib ();
        void set_c_attrib ( in C c_attrib );
    };
valuetype CState : truncatable MOFState
   supports C {
         private long a_attrib;
         private A b_attrib;
         private C c_attrib;
};
component CComponent supports C {
         attribute CState c_state;
};
home CHome manages CComponent {
         factory create_and_init_c
             ( in long a_attrib,
               in A b_attrib,
               in C c_attrib );
         factory copy_from ( in C 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 17 were used.

Example (4) Let's assume, the multiplicity for **b_attrib** of class **B** in Example (3) has the values upper>1, non-ordered and unique. According to Rule (12) , the following IDL would be generated:

```
module MyPackage {
    abstract interface B {
            ReflectiveCollection get_b_attrib ( );
            // modification is done by iterator operations
    };
};
```

Example (5) We continue with the Example (4), generating the IDL valuetype definition for **BState** as follows:

```
module MyPackage {
    valuetype BState : truncatable MOFState supports B {
            public ReflectiveCollection b_attrib; //holds a collection of AState objects
    };
};
```

6.3.2.4 Mapping of Packages

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 (27) 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 with name **format_2(<package>)**. The return type of these operations is the corresponding abstract interface derived for a package.

Rule (28) Furthermore, a component with the name **format_1 (< package identifier>)** is generated supporting the produced abstract interface. If the package is the outermost package, 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 (6) For the package MyPackage in Example (3), the following IDL definitions are produced:

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

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

6.3.2.5 Mapping of DataTypes and Enumerations

DataTypes and Enumerations are solely handled by the common mapping in Section 6.3.1.5, "Mapping of DataTypes and Enumerations," on page 20.

6.3.2.6 Mapping of Exceptions

Exceptions are solely handled by the common mapping in Section 6.3.1.6, "Mapping of Exceptions," on page 21.

6.3.2.7 Mapping of Constraints

Constraintss are solely handled by the common mapping in See Section 6.3.1.8, "Mapping of Constraints," on page 21.

6.3.2.8 Mapping of Comments

Commentss are solely handled by the common mapping in Section 6.3.1.9, "Mapping of Comments," on page 22.

6.3.2.9 Mapping of Tags

As stated earlier, tags are used to denote services provided by the generated components. Tags for standardized MOF Services such as Reflection need not to be explicitly attached to model elements.

Rule (29) Standardized tags for the IDL mapping define standardized service interfaces. These interfaces are supported through facets at the component. If a non-standardized tag is attached to a model element, the component generated for that element is extended to provide a facet of the type denoted by the tag value. The name of the tag is mapped to the name of the facet.

NOTE: It is the responsibility of the modeler to provide valid IDL identifiers in the name and value attributes of the tag. The facet type denoted by the tag value must be fully qualified.

6.3.2.10 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 (30) If a class or package in an EMOF 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::CCMReflective::RefCCMObject** in case of a class,
- **EMOF::CCMReflective::RefCCMPackage** 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 (31) Furthermore, every derived home interface is a specialization of the reflective interface **EMOF::CCMReflective::RefCCMHome**.

Example (7) If the class C and the package MyPackage of Example (3) shall support Reflection, the following IDL definitions will be generated:

```
module  MyPackage {

        component AComponent supports A {
            attribute AState a_state;
            provides EMOF::CCMReflective::RefCCMObject reflective;
    };

    home AHome : EMOF::CCMReflective::RefCCMHome
            manages AComponent
    {
      //...
    };

    abstract interface MyPackagePackage {
            AHome a();
            BHome b();
            CHome c();
    };
    component MyPackage supports MyPackagePackage {
        provides EMOF::CCMReflective::RefBaseObject reflective;
    };

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

6.3.2.11 Support for Notification-Based Model Change Communication

The former MOF 1.4 mapping implies 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 (5).

Rule (32) 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 (33) 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 (5).

For specification details, see chapter 7.

Example (8) Assume, that the class **C** of Example (3) 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 C : A, B {
        C get_c_attrib ();
        void set_c_attrib ( in C c_attrib );
    };

    valuetype CState supports C {
        private long a_attrib;
        private A b_attrib;
        private C c_attrib;
    };

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

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

6.3.2.12 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 (34) 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 (28). Both operations obtain one **in** parameter: **boolean as_value** and a return type of ReflectiveCollection. The semantic is the same as for the **all_of_*** operations in MOF1.4.

6.3.3 EMOF 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.

6.3.3.1 Mapping of Packages

The package mapping is similar to MOF1.4. 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 6.3.3.2, "Mapping of Classes," on page 31):

Rule (35) For the outermost package, an IDL interface definition is being generated with identifier **concatenate (format_1 (<package identifier>), "Factory")**. This interface contains one operation named **concatenate ("create_", format_2 (<package identifier>))** with return type of the package interface created by Rule (36).

Rule (36) Additionally, a package instance interface is derived with name **format_1 (<package identifier>)**, but without attributes for classes and associations.

Rule (37) 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 and name **concatenate (format_2 (<package identifier>), "_query")**.

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.

6.3.3.2 Mapping of Classes

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 (4) and Rule (5).

> **Rule (38)** If the class is not abstract, an IDL interface definition is being generated in the same module as the abstract interface (cp. Rule (4)) with the name **concatenate(format_1 (<class identifier>), “Concrete“)**. This interface is declared to inherit from the generated common (abstract) interface for the class and from all other concrete interfaces generated for its super classes.
>
> **Rule (39)** To access the instances state, an additional operation is generated into the non-abstract interface of Rule (38) with identifier **concatenate ("get_", concatenate (format_2(<class_name>) , "_state"))** typed to the valuetype generated according to Rule (5).

The get-operation can be used to access the instance’s state. The semantic of the attribute’s update is not trivial and is discussed in Section 7.7, “Merging of repository instances,” on page 59.

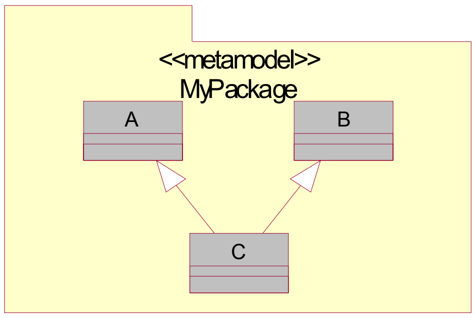
In the MOF 1.4 IDL mapping, class proxy interfaces are generated, that are used to create and query class’s instances. In this specification, we drop the class proxy interface and generate the creation operation(s) for instances into the package interface that is derived according to Rule (36). The query operations are located in the package query interface.

> **Rule (40)** 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** operation is derived analogous to Rule (25) and Rule (26). Moreover, the package query interface generated by rule Rule (37) has two operations named **concatenate ("all_of_type_", format_2 (<class identifier>))** and **concatenate ("all_of_class_", format_2 (<class identifier>))** analogous to those of Rule (34).

6.3.3.3 Mapping of Inheritance between Classes

Inheritance of classes is realized by the inheritance of the derived abstract interfaces (Section 6.3.1.3, “Mapping of Inheritance between Classes,” on page 18) and additionally via the instance interfaces (Rule (38)).

Example (9) 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 A : MOFObject {
      AState get_a_state();
    };
    abstract interface B : MOFObject {
      BState get_b_state();
    };
    abstract interface C
      : A, B {
      CState get_c_state();
    };
    valuetype AState : truncatable MOFState
      supports A { };
    valuetype BState : truncatable MOFState
      supports B { };
    valuetype CState : truncatable MOFState
      supports C { };

    // Base-IDL Mapping:
    interface MyPackageFactory {
       MyPackage create_my_package();
    };

    interface MyPackage  {
       AConcrete create_a();
       BConcrete create_b();
       CConcrete create_c();
    };
    interface AConcrete : A { };
    interface BConcrete : B { };
    interface CConcrete : C, AConcrete, BConcrete { };
} ;
```

6.3.3.4 Mapping of Attributes and Operations

Attributes and Operations are solely handled by the common mapping in Section 6.3.1.4, “Mapping of Attributes and Operations,” on page 18. Mapping of DataTypes and Enumerations

6.3.3.5 Mapping of DataTypes and Enumerations

DataTypes and Enumerations are solely handled by the common mapping in Section 6.3.1.5, “Mapping of DataTypes and Enumerations,” on page 20.

6.3.3.6 Mapping of Exceptions

Exceptions are solely handled by the common mapping in Section 6.3.1.6, “Mapping of Exceptions,” on page 21.

6.3.3.7 Mapping of Tags

As stated earlier, tags are used to denote services provided by the generated components. Tags for standardized MOF Services such as Reflection need not to be explicitly attached to model elements.

Rule (41) Standardized tags for the IDL mapping define standardized service interfaces. These interfaces are inherited by the concrete interfaces derived for the element. If a non-standardized tag is attached to a model element, the concrete interface for that element is extended to inherit from that denoted by the tag value. The name of the tag is not mapped.

NOTE: It is the responsibility of the modeler to provide valid IDL identifiers in the value attribute of the tag. The interface denoted by the tag value must be fully qualified.

6.3.3.8 Mapping of Constraints

Exceptions are solely handled by the common mapping in Section 6.3.1.8, "Mapping of Constraints," on page 21.

6.3.3.9 Mapping of Comments

Comments are solely handled by the common mapping in Section 6.3.1.9, "Mapping of Comments," on page 22.

6.3.3.10 Support for the Reflection Service

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 (42) If a class/package in an EMOF model shall support the Reflective API, an inheritance relation is added to the instance interface definition for that class (cp. Rule (38)) as for every other service. The type of the reflective interface is:

- **EMOF::Reflective::RefObject** in case of a class,
- **EMOF::Reflective::RefPackage** in case of a package and
- **EMOF::Reflective::RefFactory** in case of a package factory.

6.3.3.11 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.

6.4 CMOF Mapping Rules

Due to the fact that CMOF is constructed through package merge of the EMOF package, EMOF mapping rules are also applicable for CMOF. Thus, this section will mostly refer to these mapping rules and only introduce those specific to CMOF.

6.4.1 CMOF Common Mapping Rules

6.4.1.1 Mapping of Packages

Packages are mapped same as in EMOF (see Section 6.3.1.1, "Mapping of Packages," on page 17).

6.4.1.2 Mapping of Classes

Classes are mapped same as in EMOF (see Section 6.3.1.2, "Mapping of Classes," on page 17).

6.4.1.3 Mapping of Inheritance between Classes

Inheritance is handled same as in EMOF (see Section 6.3.1.3, "Mapping of Inheritance between Classes," on page 18).

6.4.1.4 Mapping of Attributes and Operations

In addition to the rules outlined for the EMOF common mapping in Section 6.3.1.4, "Mapping of Attributes and Operations," on page 18, the following rule does apply:

Rule (43) If an instance of class Property has a property set to one of the following, the mapping is altered as described:

- **{isDerivedUnion = true}** : if a property is a derived union, then 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 computational semantic of the **get** and **set** operations being equivalent to the UML2 abstract semantics. Note that this implies that the (sub-) set of instances for this property is included in the returned set of instances if an operation for a derived union exists (in a superclass) for which this property is a subset.
- **{redefinedProperty->size() > 0}** : redefinition of an element is covered in Section 6.4.1.5, "Mapping for Redefinitions," on page 34.

6.4.1.5 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 (44) If a property redefines another property, then one of the following applies:

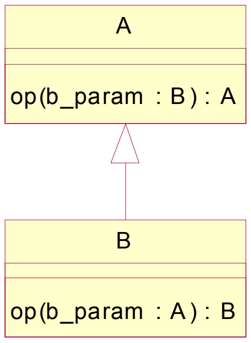
- If only the name is redefined by a property, new mutator operations are derived into the abstract interface with the redefined name as identifier in format_1 (according to Rule (9) or Rule (12)). The computational semantic of these newly declared operations is defined as a delegation to the redefined operations. The orignial operations should not be used and raise an exception when called in the more specific context.
- If only the type of a property is redefined, no additional operations are generated. Instead, the computational semantic of the operation (of Rule (9) or Rule (12)) 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

type. The "old" (inherited) operations should be made unavailable by raising an exception.

Rule (45) Operations can be viewed in the same conceptual line. Therefore, the same as in Rule (44) 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 at invocation time, it can expect (precondition) the specific type orignially defined. 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 (46) 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

Example (10) 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 A : MOFObject
{
    // the implementation of op can expect
    // that b_param is of type B
    A op ( in A b_param );
};

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

6.4.1.6 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 6.4.1.4, “Mapping of Attributes and Operations,” on page 34 are reused for navigable assocation ends.

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

Rule (47) 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 55.

Rule (48) 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 (4). Furthermore, a **typedef sequence** of type **<association-name>Link** with name **<association-name>LinkSet** for the link-set is also generated.

Rule (49) 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 (47). This valuetype contains one private member of type of the link-set's collection type (cp. Rule (48)), i.e. the sequence of derived link structs **<association_name>LinkSet**. The name for this member is **concatenate(format_2 (<association_name>), "_links")**.

Rule (50) If an association specializes another, this is mapped to inheritance of the abstract interfaces generated in Rule (47). The operations in the sub-association are only declared for those association ends that have a different name.

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.

Navigability and owned Properties

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

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 (47)) 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.

6.4.1.7 Mapping of DataTypes and Enumerations

DataTypes and Enumerations are handled same as in EMOF (see Section 6.3.1.5, “Mapping of DataTypes and Enumerations,” on page 20).

6.4.1.8 Mapping of Exceptions

Exceptions are handled same as in EMOF (see Section 6.3.1.6, “Mapping of Exceptions,” on page 21).

6.4.1.9 Mapping of Reflection

Reflection is regarded as MOF Service (orthogonal to any meta-model) and thus mapped as described in Section 6.4.2.7, “Support for the Reflection Service,” on page 38.

6.4.1.10 Mapping of Constraints

See Section 6.3.1.8, “Mapping of Constraints,” on page 21.

6.4.1.11 Mapping of Comments

Comments are handled same as in EMOF (see Section 6.3.1.9, “Mapping of Comments,” on page 22).

6.4.1.12 Mapping of Tags

Comments are handled same as in EMOF (see Section 6.3.1.11, “Mapping of Tags,” on page 22).

6.4.1.13 Mapping of Expressions and OpaqueExpressions

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.

6.4.1.14 Mapping of Extents and URIExtends

See Section 6.3.1.10, “Mapping of Extents and URIExtents,” on page 22

6.4.2 CMOF CCM Mapping Rules

6.4.2.1 Mapping of Packages

In addition to the EMOF package common mapping rules (see Section 6.3.2.4, “Mapping of Packages,” on page 26), the following rule applies:

> **Rule (51)** The package's abstract interface derived by Rule (27) contains operations for each owned association with name **format_2(<association identifier>)**. The return type of these operations is the corresponding abstract interface derived for an association (see Rule (47)).

6.4.2.2 Mapping of Classes

Classes are mapped same as in EMOF (see Section 6.3.2.1, “Mapping of Classes,” on page 22).

6.4.2.3 Mapping of Inheritance between Classes

Inheritance is handled same as in EMOF (see Section 6.3.2.2, “Mapping of Inheritance of between Classes,” on page 23).

6.4.2.4 Mapping of Attributes and Operations

In addition to the rules defined in the common mapping in Section 6.4.1.4, “Mapping of Attributes and Operations,” on page 34, the EMOF rules in Section 6.3.2.3, “Mapping of Attributes and Operations,” on page 23 do apply.

6.4.2.5 Mapping of Exceptions

Exceptions are mapped same as in EMOF (see Section 6.3.2.6, “Mapping of Exceptions,” on page 27).

6.4.2.6 Mapping of DataTypes and Enumerations

DataTypes and Enumerations are mapped same as in EMOF (see Section 6.3.2.5, “Mapping of DataTypes and Enumerations,” on page 27).

6.4.2.7 Support for the Reflection Service

Same as in the EMOF mapping, Reflection is supported via facets of components.

Rule (52) If a class or package in an EMOF 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

- **CMOF::CCMReflective::RefCCMObject** in case of a class,
- **CMOF::CCMReflective::RefCCMPackage** 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 (53) Furthermore, every derived home interface also supports the reflective interface **CMOF::CCMReflective::RefCCMHome**.

6.4.2.8 Mapping of Constraints

See Section 6.3.2.7, "Mapping of Constraints," on page 27.

6.4.2.9 Mapping of Comments

Comments are handled same as in EMOF (see Section 6.3.2.8, "Mapping of Comments," on page 27).

6.4.2.10 Mapping of Tags

Tags are handled same as in EMOF (see Section 6.3.2.9, "Mapping of Tags," on page 28).

6.4.2.11 Support for Notification-Based Model Change Communication

See Section 6.3.2.11, "Support for Notification-Based Model Change Communication," on page 29.

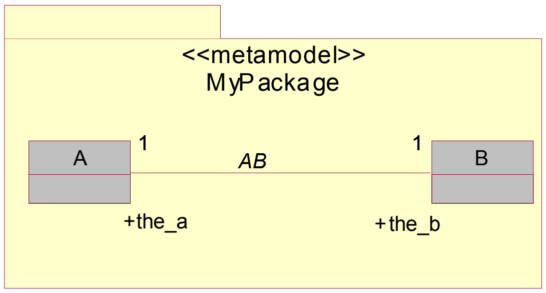
6.4.2.12 Mapping of Associations

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 6.4.1.6, "Mapping of Associations," on page 35.

Rule (54) If an association is mapped to IDL, a component declaration with the name **concatenate(format_1 (<association identifier>), "Component")** is being generated, which declares to support the abstract interface produced by Rule (47).

For specification details, see Section 7.6.3, "CMOF::Association," on page 55.

Example (11) 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 A : MOFObject {};
  abstract interface B : MOFObject {};

  struct ABLink {
     A the_a;
     B the_b;
  };
  typedef sequence < ABLink > ABLinkSet;

  abstract interface AB : MOFObject {
    ABLinkSet 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);
    B linked_objects_the_a( in A the_a )
         raises (MofError);
    A linked_objects_the_b( in B the_b )
         raises (MofError);
  };

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

  component ABComponent supports AB {
     attribute ABState ab_state;
  };

};
```

6.4.3 CMOF Base-IDL Mapping Rules

6.4.3.1 Mapping of Packages

Packages are mapped same as in EMOF (see Section 6.3.3.1, “Mapping of Packages,” on page 30).

6.4.3.2 Mapping of Classes

Classes are mapped same as in EMOF (see Section 6.3.3.2, “Mapping of Classes,” on page 31).

6.4.3.3 Mapping of Inheritance between Classes

Inheritance is handled same as in EMOF (see Section 6.3.3.3, “Mapping of Inheritance between Classes,” on page 31).

6.4.3.4 Mapping of Attributes and Operations

In addition to the rules defined in the common mapping in Section 6.4.1.4, “Mapping of Attributes and Operations,” on page 34, the EMOF rules in Section 6.3.3.4, “Mapping of Attributes and Operations,” on page 32 do apply.

6.4.3.5 Mapping of DataTypes and Enumerations

DataTypes and Enumerations are mapped same as in EMOF (see Section , “Attributes and Operations are solely handled by the common mapping in Section 6.3.1.4, “Mapping of Attributes and Operations,” on page 18. Mapping of DataTypes and Enumerations,” on page 32).

6.4.3.6 Mapping of Exceptions

Exceptions are mapped same as in EMOF (see Section 6.3.3.6, “Mapping of Exceptions,” on page 32).

6.4.3.7 Support for the Reflection Service

As in EMOF, reflection support in IDL is realized by inheritance of functionality into the instance interfaces.

Rule (55) If a class/package/association in a CMOF 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:

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

6.4.3.8 Mapping of Constraints

See Section 6.3.3.8, “Mapping of Constraints,” on page 33.

6.4.3.9 Mapping of Comments

Comments are handled same as in EMOF (see Section 6.3.3.9, “Mapping of Comments,” on page 33).

6.4.3.10 Mapping of Tags

Tags are handled same as in EMOF (see Section 6.3.3.7, “Mapping of Tags,” on page 33).

6.4.3.11 Support for Notification-Based Model Change Communication

See Section 6.3.3.11, “Support for Notification-Based Model Change Communication,” on page 33.

6.4.3.12 Mapping of Associations

Apart from the derivation for associations in the Common Mapping as described in Section 6.4.1.6, “Mapping of Associations,” on page 35, one more interface is derived:

Rule (56) If an association is mapped to IDL, an interface with identifier **concatenate (format_1 (<assocation_name>), "Concrete")** is generated inheriting from the abstract interface derived in Rule (47). Moreover, the query interface generated for this association's package according to Rule (36) is extended to contain an operation named **format_2 (<assocation_name>).** The return type is the abstract interface derived for the association according to Rule (47).

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

7 Specification Details

This section contains the language mapping details for the specification provided in section 5. In the following, the computational semantics of the operations outlined in chapter 6 is described.

7.1 Computational Semantics

Most of the computational semantics of the former MOF1.4 to IDL specification ([1]) is reused in this submission, with modifications considering the MOF2 abstract semantics. Comparing the IDL constructs derived in the MOF1.4-to-IDL mapping to this submission, 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 alternativ, 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 behaviour. 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 modifed state. Section 7.7 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 43).

- ***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 e.g., the usage of home finders. The implementation must guarantee that derived operations return apropriate homes that correspond to the accessed extent[1].

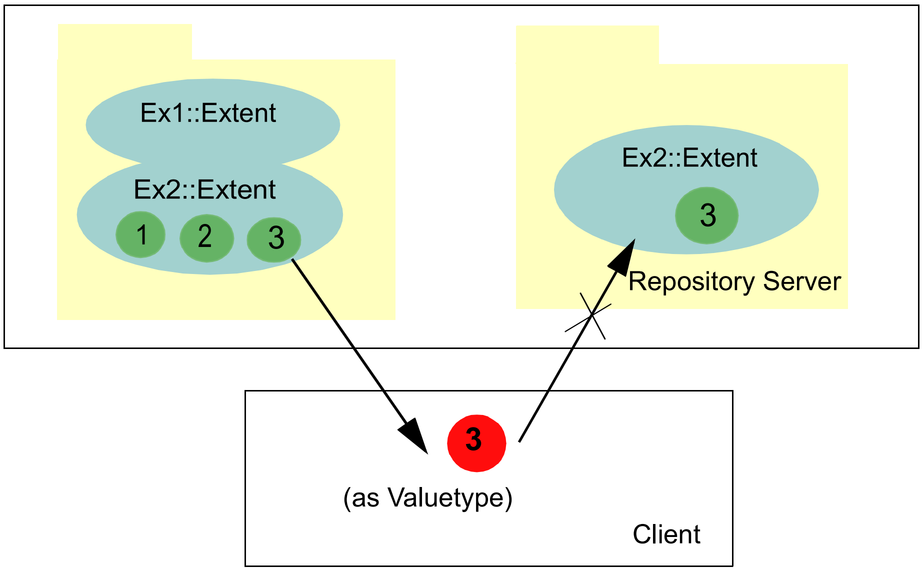
1. A vendor may choose to resolve extent-related issues via the repository ids.

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 sufficent: 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 e.g., 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 MOF1.4, we should clarify our notion of a mapping of extents regarding a realization in a CORBA environment. IDL operations/interfaces are not generated for extents for the following reason visualized the diagram below:



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.

- ***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***. This specification defines an *initial interface* for a repository implementing derived IDL 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 e.g. 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 defintions 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 equals ( in MOFObject other_object )
            raises (MofError);
     void delete () raises (MofError);
     boolean verify( in wstring ocl_expr )
            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). 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 **equals** 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 59.

Returns

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

Exception

MofError with code INVALID_OBJECT_VIOLATION is raised if the object in the repository is not available.

MOFObject::equals

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 (see also section 9.1 in [6]). 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 with code INVALID_OBJECT_VIOLATION is raised if the 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 (see also section 13.3 in [6]).

Note that references to the object are only removed automatically if the object participates in an association. In all other cases, 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 specifc 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 with code INVALID_OBJECT_VIOLATION is raised if the object in the repository 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 with code INVALID_OBJECT_VIOLATION is raised if the object in the repository 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 with code INVALID_OBJECT_VIOLATION is raised if the object in the repository 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 with code INVALID_OBJECT_VIOLATION is raised if the object in the repository is not available.

MOFObject::verify

Parameters

wstring ocl_expr: an OCL expression defining the constraint.

Semantics

Check the constraint defined by the passed OCL expression.

Returns

true, if the check was successfull, false otherwise.

Exception

MofError with code OCL_EXPRESSION_VIOLATION is raised if the OCL expression cannot be evaluated.

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 **equals** operation (see *MOFObject::equals*).

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

7.4 Exceptions

As already explained in Section 6.3.1.6, "Mapping of Exceptions," on page 21, we intend to reuse the MOF1.4 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 CLASS_CAST_VIOLATION = "org.omg.mof:structural.class_cast";
    const string OCL_EXPRESSION_VIOLATION = "org.omg.mof:structural.ocl_expression";
    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_NAVIGABLE_VIOLATION = "org.omg.mof:reflective.not_navigable";
    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.

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.

7.5.1.1 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.

7.5.1.2 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 succesful.

Exception

MofError with an appropriate code 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.

7.5.2.1 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 MOF1.4.

7.5.2.2 Derived Home interface

The Home manages the class component and corresponds to the class proxy interface of MOF1.4.

<class_name>Home::create_<class name>

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 6.2.14 and default values in 6.2.8. There exists no corresponding operation in MOF1.4.

Returns

Component instance if creation was succesful.

Exception

MofError with an appropriate code 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_<class name>

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 MOF1.4.

Returns

Component instance if creation and initialization was successful.

Exception

MofError with an appropriate code is raised if an error occurred. Note that violation of constraints may lead to exceptions.

<class_name>Home::copy_from_<class name>

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 MOF1.4. See also 7.7 for merging related issues.

Returns

Component instances managed by this home, as well as subcomponent instances.

Exception

MofError with an appropriate code is raised if an error occurred.

<class_name>Home::all_of_type

Parameters

in *<boolean>* : flag stating whether to return objects as value (TRUE) or as reference (FALSE).

Semantics

Operation to query all component instances managed by this home. It includes all subcomponent instances. Corresponds to the all_of_type_* operation in MOF 1.4.

Returns

Component instances managed by this home, including all subcomponents.

Exception

(none).

<class_name>Home::all_of_class

Parameters

in *<boolean>* : flag stating whether to return objects as value (TRUE) or as reference (FALSE).

Semantics

Operation to query all component instances managed by this home. It does not includes subcomponent instances. Corresponds to the all_of_class_* operation in MOF 1.4.

Returns

Component instances managed by this home.

Exception

(none).

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>::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 MOF1.4 (see also section 9.1 in [6]).

Returns

The property's value.

Exception

MofError with an appropriate code is raised if an error occurred. 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>::set_<prop_name>

Parameters

in *<prop_type>* new_value : the new value of the property typed to abstract interface **(*<class_name>*)**, 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 7.7. Corresponds to an attribute's s*et*-operation in MOF1.4 (see also section 9.1 in [6]).

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. WRONG_TYPE_VIOLATION is set if ***prop_type*** is not the expected one according to redefinition.

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

<class_name>::unset_<prop_name>

Parameters

(none)

Semantics

Unsets the property. Corresponds to an attribute's *unset*-operation in MOF1.4 (see also section 9.1 in [6]).

Returns

(none)

Exception

MofError with an appropriate code is raised if an error occurred.

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:

<class_name>::<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 respective classes.

Returns

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

Exception

In addition to the elements specified in *raisedExceptions,* **MofError** is included in the raise clause. The user implementation will set the MofError code appropriately.

7.5.5 EMOF::Tag

Tags are used to define service support. See Section 6.3.2.9, "Mapping of Tags," on page 28 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::Enumeration and EMOF::EnumerationLiteral

Enumerations maps to IDL enum definitions and EnumerationLiterals to identifiers within the enumeration. All details are already given in Section 6.3.1.5, "Mapping of DataTypes and Enumerations," on page 20.

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 MOF1.4 association interfaces:

<association_name>::all_<association_name>_links

Parameters

(none)

Semantics

Access to the complete link set of an association. Corresponds to *all_links*-operation of MOF1.4. If an association end is a union, all sub-association links are included in the returned link set.

Returns

<association_name>LinkSet containing all links of the association.

Exception

MofError with an appropriate code is raised if an error occurred.

<association_name>::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). Comparable to *add*-operation of MOF1.4 (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>::link_exists

Parameters

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

Semantics

Checks for whether a link is in the link-set of the assocation 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>::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>::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 the linked object (if this end has multiplicity of 1) or collection of all linked objects (if this end has multiplicity upper bound > 1) to the passed object.

Returns

<type of other end> in the case of single multiplicity ***or ReflectiveCollection*** in the case of multiple multiplicity. The found object(s) linked to passed object in the association

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 6.4.1.5, “Mapping for Redefinitions,” on page 34). Details may be found in Section 7.6.16, “CMOF::Redefinition,” on page 59.

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.

7.6.6 CMOF::Exception

The mapping for Exceptions is discussed in Section 6.3.1.6, “Mapping of Exceptions,” on page 21 and in EMOF::Operation (Section 7.5.4, “EMOF::Operation and EMOF::Parameter,” on page 54).

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 54.

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>::equals

Parameters

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

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 with code NIL_OBJECT_VIOLATION if parameter is a nil reference.

<data_type_name>::create

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 with an appropriate code if creation fails.

7.6.10 CMOF::Comment

Comments map as described in Section 6.4.1.11, "Mapping of Comments," on page 37.

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 6.4.1.10, "Mapping of Constraints," on page 36.

7.6.12 CMOF::ElementImport

The mapping for ElementImport corresponds to the mapping of AliasTypes in MOF1.4.

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.

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 6.4.1.5, “Mapping for Redefinitions,” on page 34.

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 The Reflection Service

MOF 2.0 itself defines a reflective framework which serves as a basis for the reflective API of the IDL domain (cp. [6]). The given model elements are merged with interfaces according to MOF 1.4 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 6.3.2.10, "Support for the Reflection Service," on page 28 and Section 6.3.3.10, "Support for the Reflection Service," on page 33).

In the CMOF reflective package a struct for name/value pairs is defined called RefArgument. In this mapping we use it both for EMOF and CMOF and thus defining it in the MOF Package. This is necessary to provide reflective creation of objects with automatic initialization of their properties or for reflective invocations of operations.

NOTE: The scopes of the MOF model elements (e.g. Property, Parameter) are not specified in this section. Implicitly MOF::EMOF and MOF::CMOF scopes apply.

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

typedef sequence < RefArgument > RefArgumentSet;
```

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.

8.1 EMOF Reflective

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 Base-IDL Reflection

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

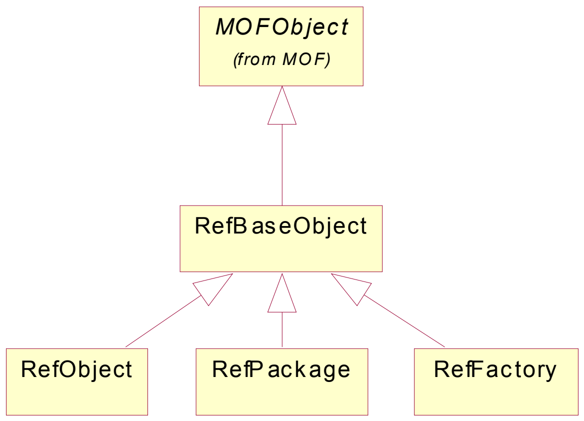


Figure 5 - Inheritance within the EMOF reflective framework

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

8.1.1.1 RefBaseObject

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

RefBaseObject::ref_get_meta_class

Parameters

(none)

Semantics

An object is returned, that holds the metamodel specifications about the instance (see also section 9.1 in [6]).

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 (see also section 9.1 in [6]).

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)

8.1.1.2 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 a 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

Property prop : 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 a ReflectiveCollection or ReflectiveSequence is returned (depending on orderedness, see also section 9.1 in [6]).

Returns

any: the value of the property. Has to be narrowed to RefObject.

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 the supplied property is not within the element.

RefObject::ref_get_by_name

Parameters

PropertyName prop : name of the property to request.

Semantics

Same as RefObject::ref_get

Returns

Same as RefObject::ref_get.

Exception

Same as RefObject::ref_get.

RefObject::ref_set

Parameters

Property 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 a ReflectiveCollection or ReflectiveSequence is expected (depending on orderedness). The collection will overwrite the old one completely (see also section 9.1 in [6]).

NOTE: In case of passing a list containing duplicates for a property requiring a unique list, 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 be a collection that is more restrictive than the desired one.

Returns

(none)

Exception

A MofError with the error_code ILLEGAL_ARGUMENT_VIOLATION is thrown, if the supplied property is not within the element.

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

If setting the value would cause the multiplicity lower bound to be violated, a MofError with code UNDERFLOW_VIOLATION is raised.

If setting the value would cause the multiplicity upper bound to be violated, a MofError with code OVERFLOW_VIOLATION is raised.

RefObject::ref_set_by_name

Parameters

PropertyName prop : name of the property to request.

any value : the new value for the property.

Semantics

Same as RefObject::ref_set.

Returns

Same as RefObject::ref_set.

Exception

Same as RefObject::ref_set.

RefObject::ref_is_set

Parameters

Property 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 (see also section 9.1 in [6]).

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.

RefObject::ref_is_set_by_name

Parameters

Property prop : name of the property to request.

Semantics

Same as RefObject::ref_is_set.

Returns

Same as RefObject::ref_is_set.

Exception

Same as RefObject::ref_is_set.

RefObject::ref_unset

Parameters

Property 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 (see also section 9.1 in [6]).

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.

If setting the value would cause the multiplicity lower bound to be violated, a MofError with code UNDERFLOW_VIOLATION is raised.

RefObject::ref_unset_by_name

Parameters

PropertyName prop : name of the property to request.

Semantics

Same as RefObject::ref_unset.

Returns

Same as RefObject::ref_unset.

Exception

Same as RefObject::ref_unset.

8.1.1.3 RefPackage

RefPackage::ref_package

Parameters

RefBaseObject package : the package type to retrieve.

Semantics

This operation provides the opportunity to request a subpackage object designated by ***package***.

Returns

RefPackage : the found package.

Exception

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

RefFactory::ref_all_of_class

Parameters

RefBaseObject class : class whose instances are to be retrieved.

Semantics

This operation has the same behavior as the non reflective all_of_class_* operations.

Returns

ReflectiveCollection : collection of found instances.

Exception

A MofError with error_code **ILLEGAL_ARGUMENT_VIOLATION** is raised, if ***class*** is not contained in the package.

RefFactory::ref_all_of_type

Parameters

RefBaseObject class : class whose instances and those of its subclasses are to be retrieved.

Semantics

This operation has the same behavior as the non reflective all_of_type_* operations.

Returns

ReflectiveCollection : collection of found instances.

Exception

A MofError with error_code **ILLEGAL_ARGUMENT_VIOLATION** is raised, if ***class*** is not contained in the package.

In MOF 1.4 this interface acts as a factory. Its 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)

8.1.1.4 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 defined in section 9.2 of [6].

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 empty reflective collection or sequence is set.

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.

RefFactory::ref_create_and_init

Parameters

RefBaseObject class : class to build the new instance from.

RefArgumentSet arguments : attribute values used to instantiate new object.

Semantics

Creates an instance of the class ***class*** initialized with the values supplied through ***arguments***.

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 or at least one of the supplied arguments does not denote an attribute of the class.

8.1.2 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 thined 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.

8.1.2.1 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 );
};
```

8.1.2.2 RefCCMPackage

The CCM reflective framework defines the RefCCMPackage as the base interface for all packages. It has no operations.

```
interface RefCCMPackage: RefCCMBaseObject {
};
```

8.1.2.3 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).

```
RefCCMObject create ();
```

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

```
RefCCMObject ref_create_and_init ( in RefArgumentSet arguments );
```

8.1.2.4 RefCCMObject

As *RefObject* from the Base-IDL mapping the *RefCCMObject* interface provides operations to change a component's properties. The syntax and semantic is completly 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);
};
```

8.2 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.

8.2.1 Base-IDL Reflection

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

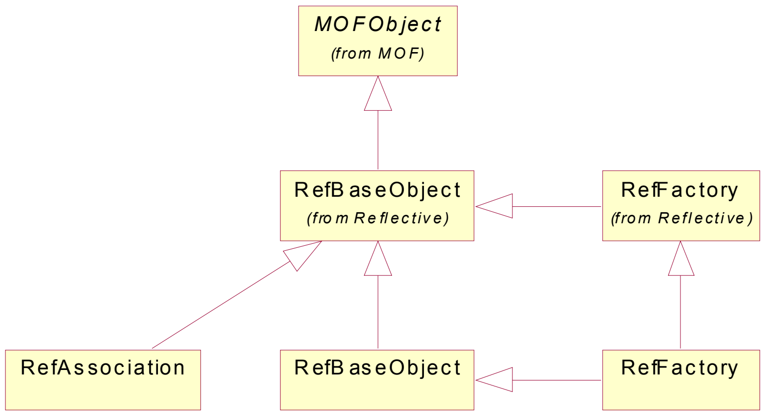


Figure 6 - Inheritance between the CMOF and EMOF reflective framework

8.2.1.1 RefBaseObject

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

In IDL an operation name is expressed is expressed with the following typedef:

typedef wstring OperationName;

RefBaseObject::ref_invoke

Parameters

Operation operation : name of the operation to invoke.

RefArgumentSet arguments : a collection of name/value pairs of type RefArgument, which represents the arguments of the operation through the name of the RefArguments and their value.

Semantics

Calls the method specified by its name. This **RefArgumentSet** 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

any : an Any value 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 such operation.

RefBaseObject::ref_invoke_by_name

Parameters

OperationName operation : name of the operation to invoke.

RefArgumentSet arguments : a collection of name/value pairs of type RefArgument, which represents the arguments of the operation through the name of the RefArguments and their value.

Semantics

Same as for ref_invoke.

Returns

Same as for ref_invoke.

Exception

Same as for ref_invoke.

8.2.1.2 RefAssociation

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

A RefLink struct for representing instances of associations is used.

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

typedef sequence < RefLink > RefLinkSet;
```

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 55.

Returns

RefLinkSet result : a collection of the set of RefLinks.

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 trough 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 55.

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 55.

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 55.

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.

RefAssociation::ref_linked_objects

Parameters

RefBaseObject end : the object, whose linked object are to be returned.

Semantics

Retrieves the objects that are linked with ***end*** in this association. The semantic is the same as the corresponding operation derived for the non reflective interfaces.

Returns

ReflectiveCollection : the collection of objects linked to ***end***.

Exception

Raises MofError with code **ILLEGAL_ARGUMENT_VIOLATION** if ***end*** is not an object that can take part in the association.

8.2.1.3 RefPackage

The CMOF RefPackage inherits from CMOF::Reflective::RefBaseObject and EMOF::Reflective::RefPackage and provides an operation to retrieves associations in the package.

RefPackage::ref_association

Parameters

RefBaseObject association : the association type to retrieve.

Semantics

This operation provides the opportunity to request the association object designated by ***association***.

Returns

RefAssociation : the found association.

Exception

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

8.2.2 CCM Reflection

The CMOF CCM Reflection defines **CCMBaseObject**, **CCMObject** and **CCMHome** as specializations of their EMOF counterparts. Moreover, the **RefCCMAssociation** is defined for associations

8.2.2.1 RefCCMAssociations

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

8.3 Reflective IDL (complete)

```
module MOF {


    struct RefArgument {
        wstring name;
        any element;
    };


    typedef sequence < RefArgument > RefArgumentSet;


module EMOF {
module Reflective {

    typedef wstring PropertyName;

    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_unset ( in Property prop ) raises (MofError);

        any ref_get_by_name ( in PropertyName prop ) raises (MofError, NotSet);
        void ref_set_by_name ( in PropertyName prop, in any new_value ) raises (MofError);
        boolean ref_is_set_by_name ( in PropertyName prop ) raises (MofError);
        void ref_unset_by_name ( in PropertyName prop ) raises (MofError);

    };

    interface RefPackage : RefBaseObject {
        RefFactory ref_factory ();
        ReflectiveCollection ref_all_of_class(in RefBaseObject class);
        ReflectiveCollection ref_all_of_type(in RefBaseObject class);
    };

    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);
        RefBaseObject ref_create_and_init ( in RefBaseObject class, in RefArgumentSet arguments)
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_unset ( in Property prop ) raises (MofError);

        any ref_get_by_name ( in Property prop ) raises (MofError, NotSet);
        void ref_set_by_name ( in Property prop, in any new_value ) raises (MofError);
        boolean ref_is_set_by_name ( in Property prop ) raises (MofError);
        void ref_unset_by_name ( in Property prop ) raises (MofError);

    };
    interface RefCCMHome : Components::CCMHome {
        RefCCMObject ref_create ();
    };

}; // CCMReflective
}; // EMOF

module CMOF {
module Reflective {

    typedef wstring Operation;
    interface RefBaseObject;

    struct RefLink {
        RefBaseObject firstObject;
        RefBaseObject secondObject;
    };

    typedef sequence < RefLink > RefLinkSet;
    typedef wstring OperationName

    interface RefBaseObject : MOF::EMOF::Reflective::RefBaseObject {
        any ref_invoke ( in Operation op,
                in RefArgumentSet arguments ) raises (MofError);
        any ref_invoke_by_name ( in OperationName op,
                in RefArgumentSet arguments ) raises (MofError);

    };

    interface RefAssociation : RefBaseObject {
        RefLinkSet 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);
        ReflectiveCollection ref_linked_objects ( in RefBaseObject end) raises (MofError);
    };

    interface RefFactory : RefBaseObject, ::MOF::EMOF::Reflective::RefFactory {
    };

}; // Reflective
module CCMReflective {

    interface RefCCMBaseObject;

    struct RefLink {
        RefCCMBaseObject firstObject;
        RefCCMBaseObject secondObject;
    };
    typedef sequence < RefLink > RefLinkSet;

    interface RefCCMBaseObject : MOF::EMOF::CCMReflective::RefCCMBaseObject {
    };
    interface RefCCMAssociation : RefCCMBaseObject,
      MOF::EMOF::CCMReflective::RefCCMObject {
        RefLinkSet 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);
        ReflectiveCollection ref_linked_objects ( in RefCCMBaseObject end) raises (MofError);
    };
    interface RefCCMHome : EMOF::CCMReflective::RefCCMHome {
            RefCCMObject ref_create_and_init ( in RefArgumentSet arguments )
                            raises (MofError);
    };

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

A 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 Specification, OMG document ptc/04-10-15

[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