Date: July 2005

UML Profile for CORBA Components

OMG Available Specification

Version 1.0

formal/05-07-06

OBJECT MANAGEMENT GROUP

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

LICENSES

PATENTS

GENERAL USE RESTRICTIONS

DISCLAIMER OF WARRANTY

OMG's Issue Reporting Procedure

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

Table of Contents

1 Scope

The CORBA Component Model (CCM) is a comprehensive component architecture based on the reliable and well-proven CORBA middleware. It contains concepts that allow multi-interface components, event based communication, port based configuration, and flexible implementation structures. These concepts are specified in the CCM metamodel defined in the OMG CORBA Components Specification, formal/02-06-65, located at this URL: http://www.omg.org/cgi-bin/doc?formal/02-06-65.

This specification provides the normative UML Profile for CORBA Components. The Profile defines a set of UML 1.5 extensions to represent CCM concepts like CORBA Component or CORBA Home. It is based on the UML Profile for CORBA, formal/02-04-01 (http://www.omg.org/cgi-bin/doc?formal/02-04-01) and extends this Profile to allow the modeling of additional concepts of ComponentIDL and Component Implementation Framework (CIF) according to the CCM metamodel.

This specification is compliant with the Model Driven Architecture (MDA) defined by the OMG and provides a standard means for expressing CCM-based applications (PSMs) using UML notation and thus to support all kind of MDA model transformations such as PIM→PSM or PSM→PSM and also work with MOF repositories.

2 Conformance

This specification defines three conformance points. Implementations must support all these conformance points:

- Implementation of the UML Profile for CORBA defined in formal/02-04-01.
- Implementation of the ComponentIDL Profile defined in Section 8.1, “ComponentIDL Profile,” on page 8.
- Implementation of the CIF Profile defined in Section 8.2, “UML Profile for CIF,” on page 25.

3 Normative References

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

- Unified Modeling Language (UML) Specification, Version 1.5
- CORBA Components Specification, Version 3.0
- The UML Profile for CORBA, Version 1.0
- MOF Specification, Version 1.4

4 Terms and Definitions

For the purposes of this specification, the terms and definitions given in the normative references and the following apply.

Term	Definition
artifact	An element that descibes abstractions from programming language constructs like classes.
component	A basic metatype in CORBA that is a specialization and extension of an interface definition.
component type	A specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an Interface Repository.
facet	A distinct named interface provided by the component for client interaction.
factory	A home operation that supports creation semantics.
finder	A home operation that supports search semantics.
home	A metatype that acts as a manager for instances of a specified component type.
port	A surface feature through which clients and other elements of an application environment may interact with a component.
receptacle	A named connection point that describes the component's ability to use a reference supplied by some external agent.
segment	An element that describes a segmented implementation structure for a component implementation.

5 Symbols

CCM CORBA Component Model

CIF Component Implementation Framework

IDL Interface Definition Language

MDA Model Driven Architecture

PIM Platform Independent Model

PSM Platform Specific Model

UML Unified Modeling Language

6 Additional Information

6.1 Changes to Adopted OMG Specifications

There are no changes to existing OMG Specifications.

6.2 The Relationship to the UML 2.0

The CCM profile is defined as a UML 1.5 profile. In September, 2000, OMG started to work on the Release 2.0 major revision of the UML specification. UML 2.0 is tailored to MDA requirements, and is being proposed in four separate RFPs: UML Infrastructure, UML Superstructure, Object Constraint Language, and UML Diagram Interchange. It is expected that new UML 2.0 concepts (e.g., port, component, etc.) will simplify the modeling of component-based infrastructures like CCM or EJB. However, the necessity of the UML profile for CCM will not disappear and it will be a possible subject for a separate RFP in the future.

6.3 Acknowledgements

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

- Alcatel
- Fraunhofer Institute FOKUS
- IKV++ Technologies AG
- Laboratoire d'Informatique Fondamentale de Lille
- Technical University Berlin
- Thales

NOTE: The technology found in this specification is based on the work of the MASTER project (http://www.esi.es/Master) and the COACH project (http://www.ist-coach.org/) of the IST Program of the European Commission. The submitters would like to thank the participants of these projects for their contributions and review activities.

7 Overview

The UML Profile for CORBA Components specification was designed to provide a standard means for expressing the semantics of CORBA Component Model (CCM) using UML notation and thus to support expressing these semantics with UML tools. The Profiles described in this manual are aimed at software designers and developers who want to design component-based CORBA applications. There is already an OMG Standard that defines how to model pure CORBA applications using UML: The UML Profile for CORBA (or CORBA Profile, formal/02-04-01). The UML Profile for CORBA Components (or CCM Profile) is considered as an extension to the pure CORBA Profile and strictly based on its definition exactly like the CORBA Components Standard (formal/02-06-65) is considered as an extension to the Common Object Request Broker Architecture and Specification that contains the architecture and specifications for base CORBA Interface Definition Language (IDL). The dependencies between UML, CORBA Profile, and CCM Profile are shown in Figure 7.1.

7.1 UML Subset Definition

The UML Profile for CCM depends on the UML Core package (formal/03-03-01) and the UML Profile for CORBA.

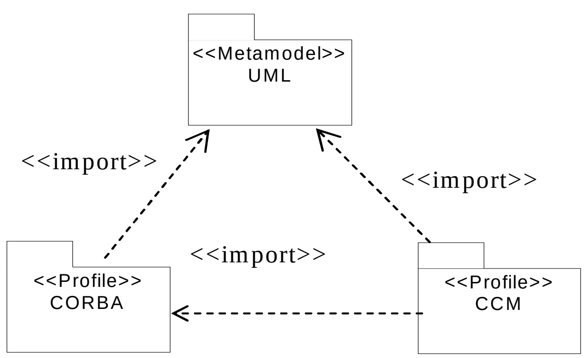

Figure 7.1 - Import dependencies between UML Metamodel, CORBA Profile and CCM Profile packages

The following concrete UML metaclasses, and implicitly all super-metaclasses of these metaclasses, are used:

- Generalization
- Association
- Class
- Operation
- Constraint
- Package

The CCM metamodel is defined on top of the CORBA (BaseIDL Package, see Figure 7.2) metamodel. We have the same relationship at the profile level, it means that the CCM Profile uses the CORBA Profile. The CCM Profile uses the definition of all CORBA data types and specializes the following stereotypes defined in the CORBA Profile:

- «CORBAInterface»
- «CORBAValue»

7.2 CCM Package structure

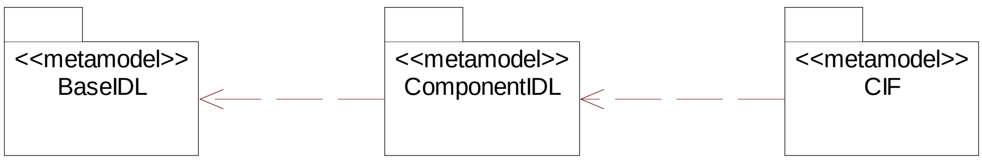

Figure 7.2 - Package structure of CCM metamodels

As shown in Figure 7.2 the whole CCM concept space is represented by three metamodel Packages: BaseIDL, ComponentIDL, and CIF (Component Implementation Framework).

The ComponentIDL Package expresses the Component Model extensions to CORBA IDL. This package is dependent upon the BaseIDL Package, which is a MOF-compliant metamodel of the base CORBA IDL. Since these extensions are derived from the previously-existing IDL base, it was not possible to define a MOF-compliant metamodel for the extensions without defining a MOF-compliant metamodel for the IDL base. Therefore, the BaseIDL metamodel was specified and explained in the CORBA Components Specification, section 8.1.1 “BaseIDL Package.” The UML Profile for CORBA is based on this metamodel. The ComponentIDL metamodel concepts are specified and explained in the CORBA Components Specification, section 8.1.2 “ComponentIDL Package.” This document gives only the short introduction of the ComponentIDL Package in Section 8.1.1, “ComponentIDL metamodel,” on page 8.

The CIF Package contains metaclasses and associations for definition of the programming model for constructing component implementations. This CIF Package depends on the ComponentIDL Package since its main purpose is to enable the modeling of implementations for components specified using the ComponentIDL definitions. The short introduction of the CIF Package content is given in Section 8.2.1, “CIF Metamodel,” on page 25.

The XMI format for the exchange of CCM metadata is provided in the CORBA Components Specification, sections 8.3.1 (XMI DTD for the Interface Repository Model) and 9.4.1 (XMI DTD for the CIF Metamodel).

8 CCM Profile Definition

The general definition of a UML Profile can be found in the UML Profile for CORBA specification in section 2.1 "General Definition of a UML Profile."

The CCM Profile specifies a set of UML extensions like stereotypes, tagged values, and constraints. The concept of stereotype is the most important and provides a way of classifying elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs. The classified element properties can be expressed via tagged values. For the graphical representation of the "virtual" metamodel we use the following approach:

- The model is expressed via UML class diagrams.
- Each stereotype is expressed via a stereotyped with <<stereotype>> Classifier box.
- Each tagged value is expressed via comma delimited sequence of property specifications inside a pair of braces ({ }) by a stereotype.
- Each stereotype is a client in a UML Dependency Relationship with the UML metaclass that it extends. These Dependencies are stereotyped with <<stereotype>>.
- Generalization Relationships among stereotypes are expressed in the standard UML manner.

The UML "virtual" metamodel of all stereotype and tagged value declarations for the CCM Profile is provided in the OMG documents ptc/2005-01-02 (Rose model file) or ptc/2005-01-03 (XML file).

An alternative and usually more compact way of specifying stereotypes and tags is using tables. The columns of the stereotype specification table are defined as follows:

- Stereotype: the name of the stereotype.
- Base Class: the UML metamodel element that serves as the base for the stereotype.
- Parent: the direct parent of the stereotype being defined (NB: if one exists, otherwise the symbol "NA" is used).
- Tags: a list of all tags of the tagged values that may be associated with this stereotype (or NA if none are defined).
- Description: an informal description with possible explanatory comments.

The columns of the tag specification table are defined as follows:

- Tag: the name of the tag.
- Stereotype: the name of the stereotype that owns this tag, or "NA" if it is a stand alone tag.
- Type: the name of the type of the values that can be associated with the tag.
- Multiplicity: the maximum number of values that may be associated with one tag instance.
- Description: an informal description with possible explanatory comments.

This specification provides both forms of specifying stereotypes and tagged values: tabular and graphical.

Constraints represent semantic information attached to an element. A list of constraints associated with a stereotype is expressed in English and OCL separately from the stereotypes and tags specification. The following OCL convenience operation is used in the UML Profile for CCM, it is defined in [7] for the metaclass *ModelElement* in order to produce more compact and readable OCL:

The operation *isStereotyped* determines whether the ModelElement has a Stereotype whose name is equal to the input name.

isStereotyped : (stereotypeName : String) : Boolean;
self.stereotype.name = stereotypeName

8.1 ComponentIDL Profile

8.1.1 ComponentIDL metamodel

Figure 8.1 gives the structure of the CCM external concepts. A more detailed description of these concepts can be found in the CCM OMG standard definition.

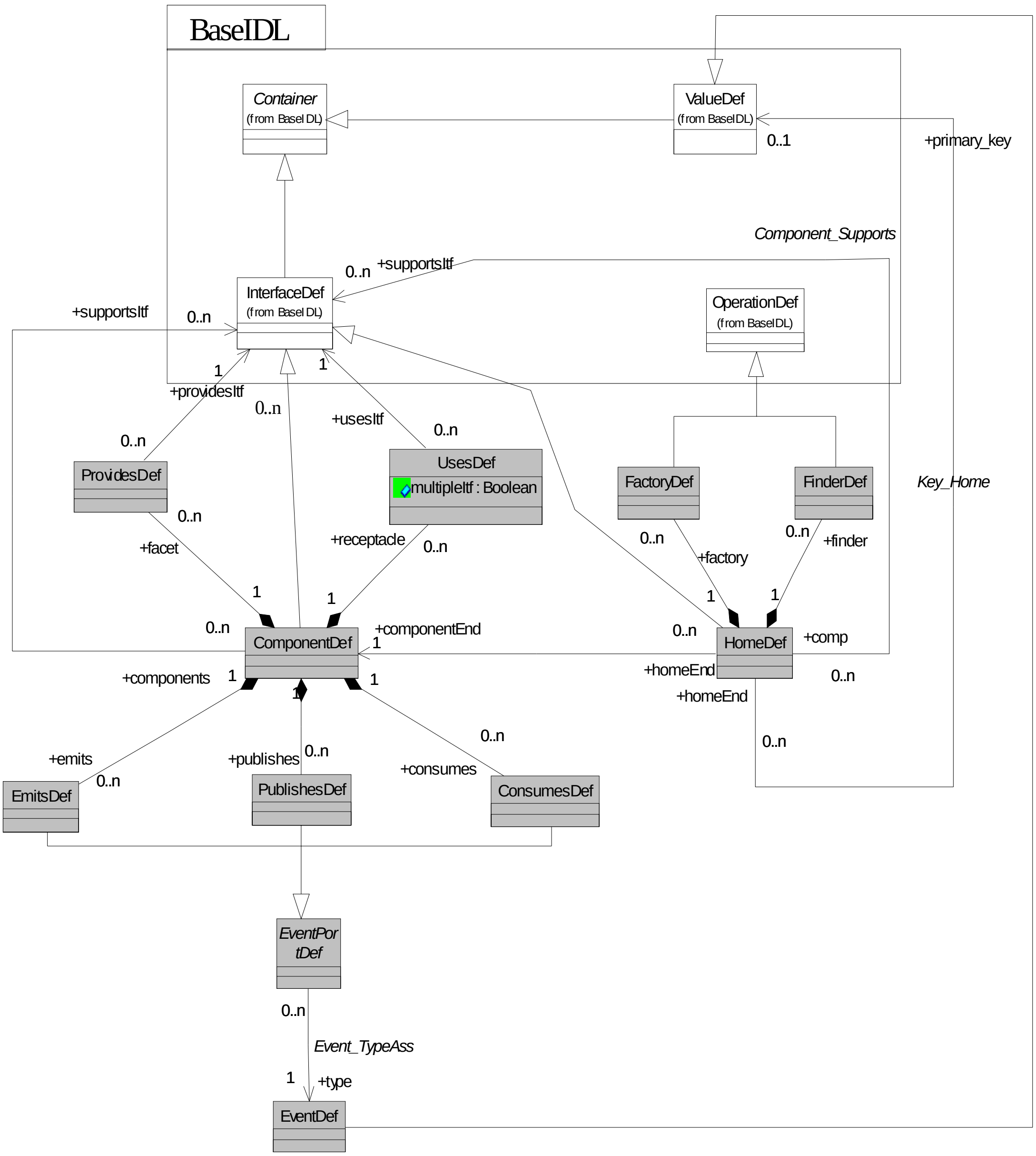

Figure 8.1 - ComponentIDL Metamodel

This abstract model is used to characterize CORBA Component interfaces. The ComponentIDL (known also as IDL3) language has been defined to describe instances of this Metamodel. It is a generalization of the OMG IDL language.

A component type defines attributes and ports. The attributes are used to configure the component. By using ports, components can use or provide a set of services (typed with a CORBA interface). There are four kinds of ports:

1. A facet is a component provided interface. It is a synchronous communication mechanism.

2. A receptacle is a component required interface. It is a synchronous communication mechanism.
3. An event sink is a component provided interface. It is an asynchronous communication mechanism.
4. An event source is a component required interface. It is an asynchronous communication mechanism.

A component is a kind of interface. A component can inherit from another one and support one or more interfaces. A component cannot inherit from several components at the same time. Multiple inheritance is only possible for interfaces.

A facet represents a component's role. It is described using an interface. A facet is the only visible part for clients. It is only a declarative part. Clients have no access to the implementation part. Facet implementations are hidden inside the component. Facets and components have the same lifecycle. Each facet has its own reference.

With a receptacle a component can use a reference. This relationship is called a connection. Connections are used for component assembly. There are two receptacle kinds. Simple receptacle can only use a single reference. "Multiple" receptacle can use several references.

The CCM also provides a provider/consumer event model. There are two kinds of event ports: event source and event sink. An event source can be either an emitter (only one consumer) or a publisher (several consumers). Event sources are used to send events; event sinks are used to receive events.

CORBA Components are managed by homes. A component home provides component factory operations. It can also provide component finder operations. "Home" supports single inheritance.

8.1.2 Profile Definition

Component

A CORBA Component is defined using a UML "CORBAComponent" stereotyped class. A "CORBAComponent" can inherit from another one (single inheritance) using the UML generalization. It can also inherit from a set of CORBA interfaces. These relationships are materialized with "CORBASupports" stereotyped generalizations.

Table 8.1 - CORBAComponent and CORBASupports Stereotypes

Stereotype	Base Class	Parent	Tags	Description
CORBAComponent <<CORBAComponent>>	Class	CORBA Interface	NA	A CORBA Component is a class with specific, named collection of features like attributes or ports. An instance of the component has state and identity.
CORBASupports <<CORBASupports>>	Generalization	NA	NA	CORBASupports is a generalization relationship between component and its inherit interface(s).

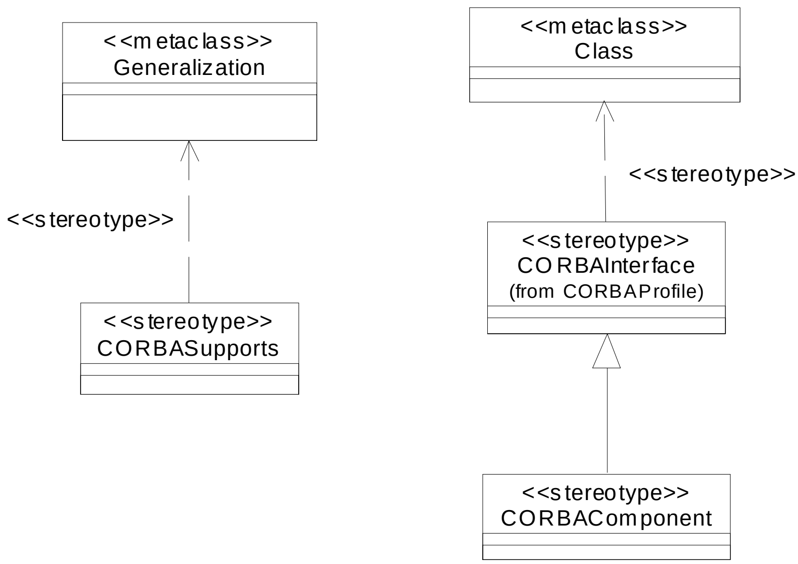

Figure 8.2 - Explicit Modeling of CORBAComponent and CORBASupports Stereotypes

<<CORBAComponent>> Constraint

A « CORBAComponent » is a kind of « CORBAInterface ». Each “CORBAComponent” must respect the “CORBAInterface” constraints. It must also respect the following additional constraints:

- A «CORBAComponent» cannot own operations:

  self.feature→forAll(not oclIsKindOf (behavioralFeature))

- A «CORBAComponent» can only inherit from a «CORBAComponent» or a «CORBAInterface»:

  self.generalization→forAll (g : Generalization | g.parent.isStereotyped ("CORBAComponent") or g.parent.isStereotyped("CORBAInterface"))

- Only single inheritance is possible between «CORBAComponent»:

  self.generalization→
  select(parent.isStereotyped("CORBAComponent"))→size <= 1

- Each «CORBAComponent» inheritance from a «CORBAInterface» must be stereotyped «CORBASupports»:

  self.generalization→forAll (g : Generalization | g.parent.isStereotyped("CORBAInterface") implies g.isStereotyped("CORBASupports"))

Example

The CCM Component inheritance model is the UML counterpart of the following IDL3 declaration:

```
interface I1 { } ;
interface I2 { } ;
component C1 supports I1, I2 { } ;
component C2 : C1 { } ;
```

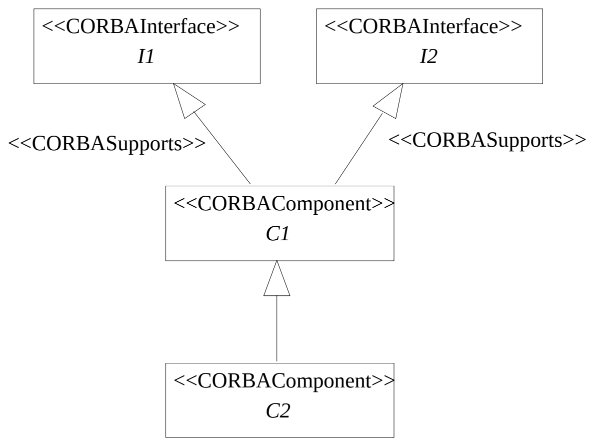

Figure 8.3 CCM Component inheritance

Facet and Receptacle

The facets are described using a composition association between a CORBAComponent and a CORBA interface. This association must be stereotyped “CORBAProvides.” The name of this stereotyped association gives the facet name. The AssociationEnd cardinalities are [1..1] for both association sides.

The receptacles are described using a composition association between a CORBAComponent and a CORBA interface. This association must be stereotyped “CORBAUses.” The role name of the interface AssociationEnd gives the receptacle name. The component side AssociationEnd cardinality must be [1..1]. The receptacle side AssociationEnd cardinality is [1..n] where n is the receptacle cardinality. Table 8.2 describes facet and receptacle stereotypes. Table 8.3 defines the associated tagged value „multiple.“

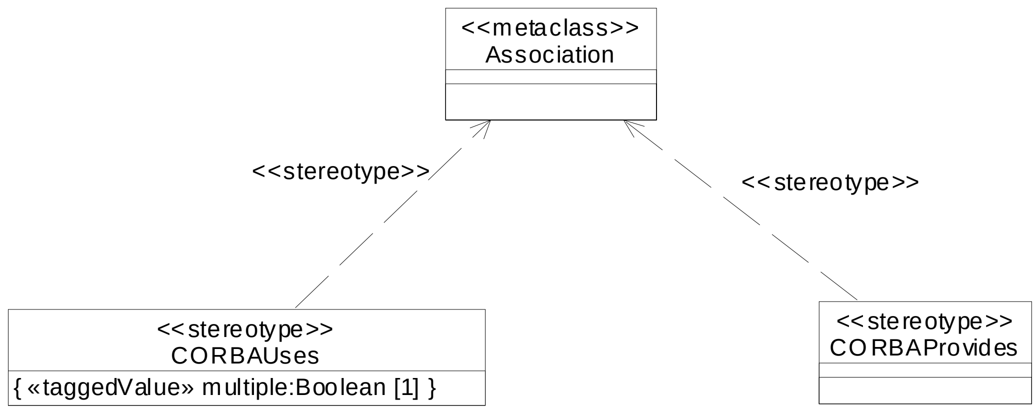

Figure 8.4 - Explicit Modeling of CORBAUses and CORBAProvides Stereotypes

Table 8.2 - CORBAProvides and CORBAUses Stereotypes

Stereotype	Base Class	Parent	Tags	Description
CORBAProvides <<CORBAProvides>>	Association	NA	NA	A CORBAProvides is an association between component and its provided interfaces that represents the component port called facet.
CORBAUses <<CORBAUses>>	Association	NA	multiple	A CORBAUses is an association between component and interfaces that component uses. The association represents the component port called receptacle.

Table 8.3 - Tag definition for CORBAUses Stereotype

Tag	Stereotype	Type	Multiplicity	Description
multiple	CORBAUses	Boolean	1	Indicates whether the multiple connections to the receptacle may exist simultaneously or not.

Constraints

- It’s an association between a «CORBAComponent» and a «CORBAInterface»:

  self.connection→exists(participant.isStereotyped("CORBAComponent")) and self.connection→exists (participant.isStereotyped("CORBAInterface"))

- The «CORBAComponent» side is a composition:

self.connection→exists(participant.isStereotyped("CORBAComponent") and aggregation = #composite)

CCMFacet additional constraints

- It's an association stereotyped «CORBAProvides»:

  self.isStereotyped("CORBAProvides")

- The "CORBAInterface" side cardinality must be 1:

  self.connection→exists(participant.isStereotyped("CORBAInterface") and multiplicity.min=1 and multiplicity.max=1)

CCMReceptacle additional constraints

- It's an association stereotyped «CORBAUses»:

  self.isStereotyped("CORBAUses")

- The «CORBAInterface» side cardinality must be 1 for simple receptacle.

  self.connection→exists(participant.isStereotyped("CORBAInterface") and multiplicity.min=1 and multiplicity.max=1)

- The «CCMReceptacle» side cardinality must be greater than one for multiple receptacles.

  self.connection→exists(participant.isStereotyped("CORBAInterface") and multiplicity.min=1 and multiplicity.max>1)

Example

The Facets and Receptacles model is the UML counterpart of the following IDL3 declaration:

```
interface  I1 { };
interface I2 { };
interface I3 { };
component C1 {
    provides I1 facet1;
    uses  I2 receptacle1;
    uses multiple I3 receptacle2;
};
```

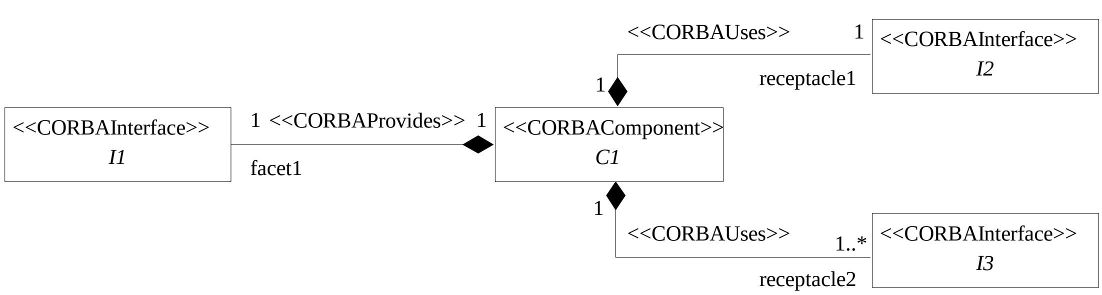

Figure 8.5 - Facets and Receptacles

Events

Event types are defined using a « CORBAEvent » stereotyped class. The “CORBAEvent” stereotype is a specialization of the « CORBAValue » stereotype. It inherits from all “CORBAValue” constraints. Event source is defined either by a “CORBAEmits” stereotyped composition association or by a “CORBAPublishes” association between a “CORBAComponent” and a “CORBAEvent.” The event source port name is defined using this stereotyped association name.

Event sinks are defined the same way using a “CORBAConsumes” stereotyped composition association.

Table 8.4 describes event and event port stereotypes.

Table 8.4 - CORBAEvent, CORBAEventPort, CORBAEmits, CORBAPublishes and CORBAConsumes Stereotypes

Stereotype	Base Class	Parent	Tags	Description
CORBAEvent <<CORBAEvent>>	Class	CORBAValue	NA	A CORBAEvent class represents an event type (data type for component current state) that one component wishes to notify (event source) another component about (event sink).
CORBAEventPort <<CORBAEventPort>>	Association	NA	NA	A CORBAEventPort is an association between a component and an event. CORBAEventPort is an abstract class and cannot be instantiated directly.
CORBAEmits <<CORBAEmits>>	Association	CORBAEventPort	NA	A CORBAEmits is an association between component and events that this component emits.
CORBAPublishes <<CORBAPublishes>>	Association	CORBAEventPort	NA	A CORBAPublishes is an association between component and events that this component publishes.
CORBAConsumes <<CORBAConsumes>>	Association	CORBAEventPort	NA	A CORBAPublishes is an association between component and events that this component consumes.

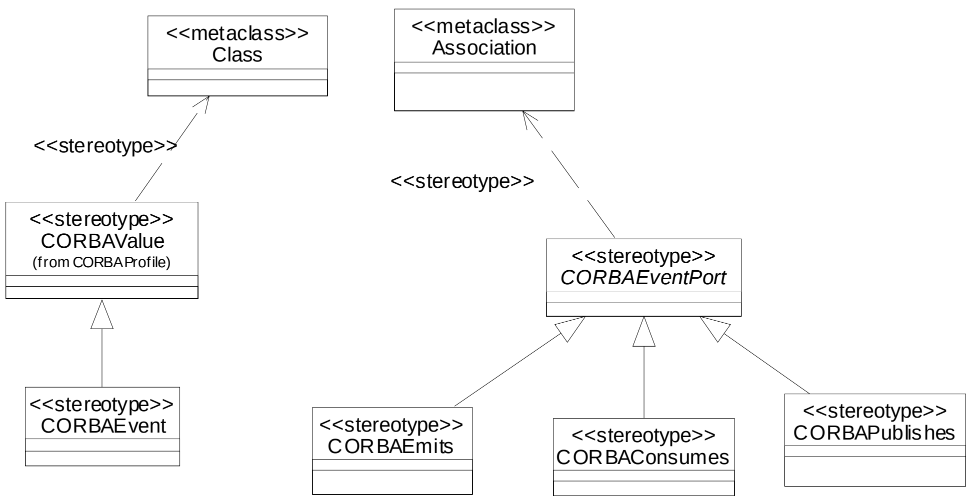

Figure 8.6 Explicit Modeling of CORBAEvent, CORBAEventPort, CORBAEmits, CORBAPublishes and CORBAConsumes Stereotypes

Constraints

- It’s a binary association.

  self.connection→size=2

- It’s an association between a «CORBAComponent» and a «CORBAEvent».

  self.connection→exists(participant.isStereotyped("CORBAComponent")) and self.connection->exists (participant.isStereotyped("CORBAEvent"))

- The « CORBAComponent » side is a composition.

  self.connection→exists(participant.isStereotyped("CORBAComponent") and aggregation = #composite)

- The «CORBAEvent» side cardinality must be 1.

  self.connection→exists(participant.isStereotyped("CORBAEvent") and multiplicity.min=1 and multiplicity.max=1)

«CORBAEmits» constraints

- The «CORBAEmits» side cardinality must be 1.

  self.connection→exists(participant.isStereotyped("CORBAEmits") and multiplicity.min=1 and multiplicity.max=1)

«CORBAPublishes» constraints

- The «CORBAPublishes» side cardinality can be 1 or more.

  self.connection→exists(participant.isStereotyped("CORBAPublishes") and multiplicity.max>1)

Example

The Event sink and event source model is the UML counterpart of the following IDL3 declaration:

```
eventtype E1 { };
eventtype E2 { };
eventtype E3 { };
component C1 {
    emits E1 source1;
    publishes E2 source2;
    consumes E3 sink1;
};
```

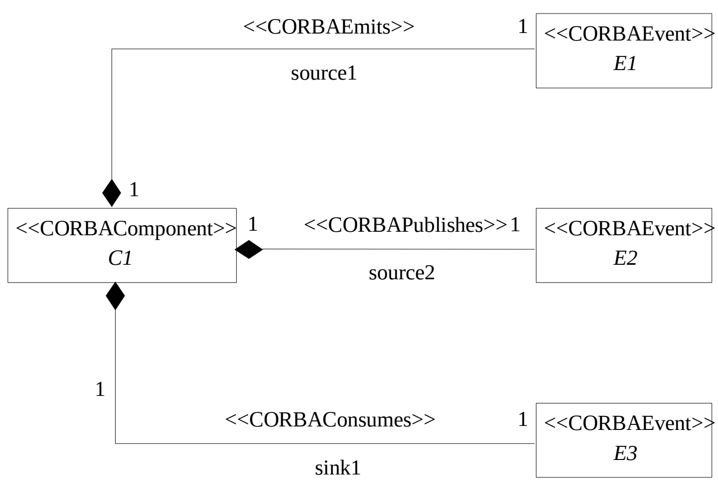

Figure 8.7 - Event sink and event source

Component Home

A Component home is described using a “CORBAHome” stereotyped class. This stereotype specializes the “CORBAInterface” stereotype. A component home must be associated to a component type. This relationship is made explicit using a “CORBAManages” stereotyped association between a “CORBAHome” and a “CORBAComponent.”

A “CORBAHome” can inherit from another “CORBAHome” (single inheritance) using a UML generalization. A “CORBAHome” can support several “CORBAInterface.” Each “CORBAInterface” generalization must be stereotyped ”CORBASupports.”

A “CORBAHome” can be associated with a primary key (necessary for persistent components). There is exactly one key instance for each (persistent component, home) instance couple. To enforce this constraint, the primary key is represented using a ”CORBAValue” stereotyped AssociationClass. The ”CORBAValue” stereotype was defined in the UML Profile for CORBA [7].

A “CORBAHome” can own attributes and operations. The stereotype “CORBAFactory” is used for the component factory operations. The stereotype “CORBAFinder” is used for components finder operations. Table 8.5 describes the component home stereotypes.

Table 8.5 - CORBAHome, CORBAFactory, CORBAFinder, CORBAManages, and CORBAPrimaryKey Stereotypes

Stereotype	Base Class	Parent	Tags	Description
CORBAHome <<CORBAHome>>	Class	CORBAInterface	NA	A CORBAHome is a class that acts as a manager for instances of a specified component. CORBA Home inherits from CORBA Interface and provides operations (factory and finder) to manage component life cycles, and optionally, to manage associations between component instances and primary key values. A home must be declared for every component declaration.
CORBAFactory <<CORBAFactory>>	Operation	NA	NA	CORBAFactory is an operation that creates a new instance of the component associated with the home object.
CORBAFinder <<CORBAFinder>>	Operation	NA	NA	A CORBAFinder is an operation that obtains homes for a particular component.
CORBAManages <<CORBAManages>>	Association	NA	NA	CORBAManages is an association between components and their homes.
CORBAPrimaryKey <<CORBAPrimaryKey>>	Association	NA	NA	CORBAPrimaryKey is an association between home and its primary key.

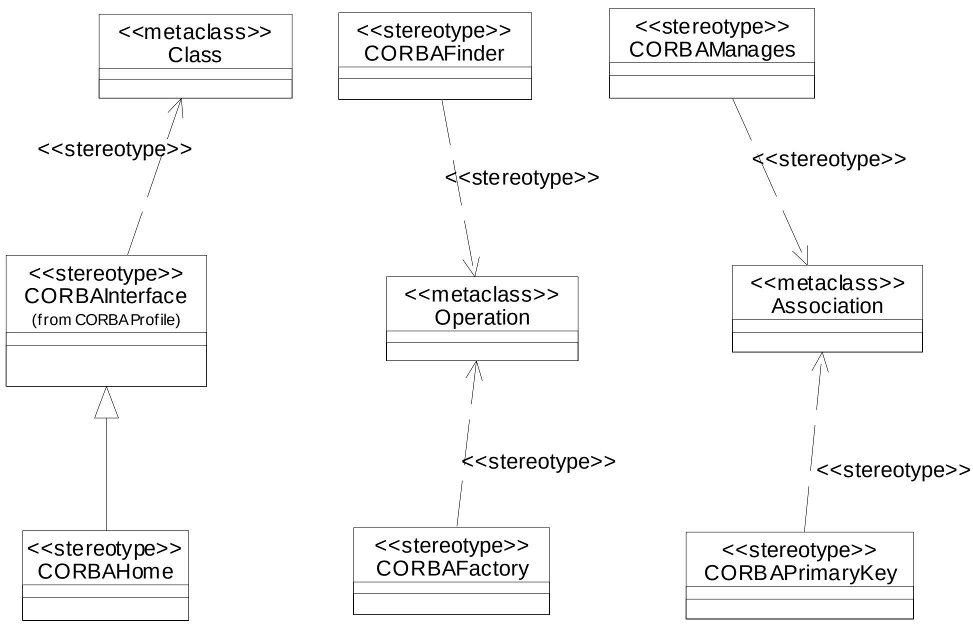

Figure 8.8 Explicit Modeling of CORBAHome, CORBAFactory, CORBAFinder, CORBAManages and CORBAPrimaryKey stereotypes

Constraints

- There is exactly one « CORBAManages » association for each Home.

  self.connection→select(isStereotyped("CORBAManages"))→size = 1

- The «CORBAHome» side cardinality must be 1..1.

  self.connection→exists(participant.isStereotyped("CORBAHome")) and multiplicity.min=1 and multiplicity.max=1)

- The «CORBAComponent» side cardinality must be "0..n."

  self.connection→exists(participant.isStereotyped("CORBAComponent")) and multiplicity.min=0 and multiplicity.max=n)

- A « CORBAHome » can inherit from one « CORBAHome » at most.

  self.generalization→select(parent.isStereotyped("CORBAHome"))→size=1

- If "CORBAHome" h_1 inherits from "CORBAHome" h_2 and h_2 manages "CORBAComponent" C_2, then h_1 must manage C_2 or any other component C_1 *that* inherits from C2.

  ***let h1=self and let h2=self.generalization→select(parent.isStereotyped("CORBAHome")) and h2→notEmpty implies let C2=h2.connection→
  select(participant.isStereotyped("CORBAComponent")) and let C1=
  h1.connection→select(participant.isStereotyped("CORBAComponent")) and (C1 = C2 or C1.allParents→includes(C2))***

- If « CORBAHome » h1 inherits from h2, and « CORBAHome » h2 is associated with primary key k2, then h1 must be associated with k2 or with a primary key k1 that inherits from k2.

let h1=self and let h2=self.generalization→select(parent.isStereotyped("CORBAHome")) and h2→notEmpty implies let k2=h2.connection
→select(isStereotyped("CORBAManages")).LinkToClass.ClassPart and let k1=self.connection
→select(isStereotyped("CORBAManages")).LinkToClass.ClassPart and (k1 = k2 or k1.allParents->includes(k2))

- Each «CORBAHome» inheritance from a «CORBAInterface» must be stereotyped.

self.generalization→forAll
(g : Generalization | g.parent.isStereotyped("CORBAInterface")
implies g.isStereotyped("CORBASupports"))

«CORBAManages» constraints

- It's an association between a «CORBAHome» and a «CORBAComponent».

self.connection→exists(participant.isStereotyped("CORBAHome")) and self.connection→exists(participant.isStereotyped("CORBAComponent"))

«CORBAValue» of a primary key constraints

- The valuetype of a primary key
  [1] Must not have private state members.
  [2] Must not have members that are interfaces.
  [3] Must have at least one state member.
  [5] Must descend directly or indirectly from Components::PrimaryKeyBase.
  [4] Constraints [1], [2], and [3] apply recursively to valuetype members that are valuetypes.

[1,2,3,4] isAcceptableKeyType(type)

```
isAcceptableKeyType(valueType : ValueDef) : boolean
{
 valueType.contents.forAll (c | c.oclIsTypeOf(ValuefMemberDef) implies
 c.OclAsType(ValueMemberDef).isPublicMember) and
 valueType.contents.forAll (not oclIsKindOf (InterfaceDef)) and
 valueType.contents.exists (oclIsTypeOf(ValueMemberDef)) and
 valueType.contents.forAll (c | c.oclIsKindOf (ValueDef) implies isAcceptableKeyType (c))
}
```

[5] type.descendsFrom("Components::PrimaryKeyBase")

```
descendsFrom(absoluteName : string) : boolean
{
 descendsFrom(absoluteName) =
 if self.absoluteName = absoluteName  then
    true
 else
   if base->isEmpty then
       false
   else
```

```
        if base.descendsFrom(absoluteName) then
          true
        else
          false
        endif
      endif
    endif
    }
```

«CORBAHomeFactory» constraints

- A « CORBAHomeFactory » operation has only input parameters.

  self. parameter→forAll(kind=#in)

- A «CORBAHomeFactory» can only be defined in a "CORBAHome."

  self.owner.isStereotyped("CORBAHome")

«CORBAHomeFinder» constraints

- A « CORBAHomeFinder » has only input parameters.

  self. parameter→forAll(kind=#in)

- A « CORBAHomeFinder » can only be defined in a "CORBAHome."

  self.owner.isStereotyped("CORBAHome")

Example

The following IDL3 example can be represented using the Component Home model.

```
module Components {
abstract valuetype PrimaryKeyBase {};
};
valuetype Key : Components::PrimaryKeyBase { public string _key; };
component C1 { };
component C2 { };
home C1Home manages C1 primarykey Key {
    finder  findByName(in string name);
    factory create(in string name);
};
interface I1 { };
interface I2 { };
home myHome supports I1, I2 manages C2 { ... };
home C2Home : myHome manages C2 { };
```

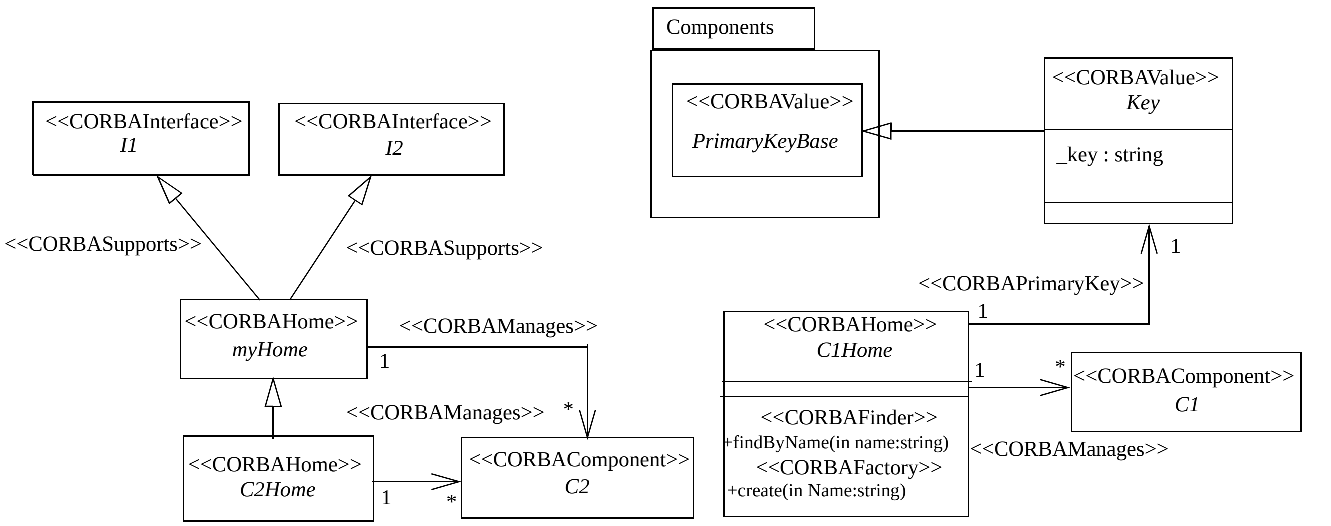

Figure 8.9 - Component Home

8.1.3 Metamodel to Profile Mapping

The mapping between the profile and the metamodel of the CCM is specified by giving the relation between the metamodel elements and the elements of the profile. It is shown in the following figures. The graphical relation "represents" means that the specific modelElement of the metamodel is represented by the associated construct(s) of the profile. For example, an instance of the metaclass ComponentDef is represented by a UML class stereotyped as CORBAComponent.

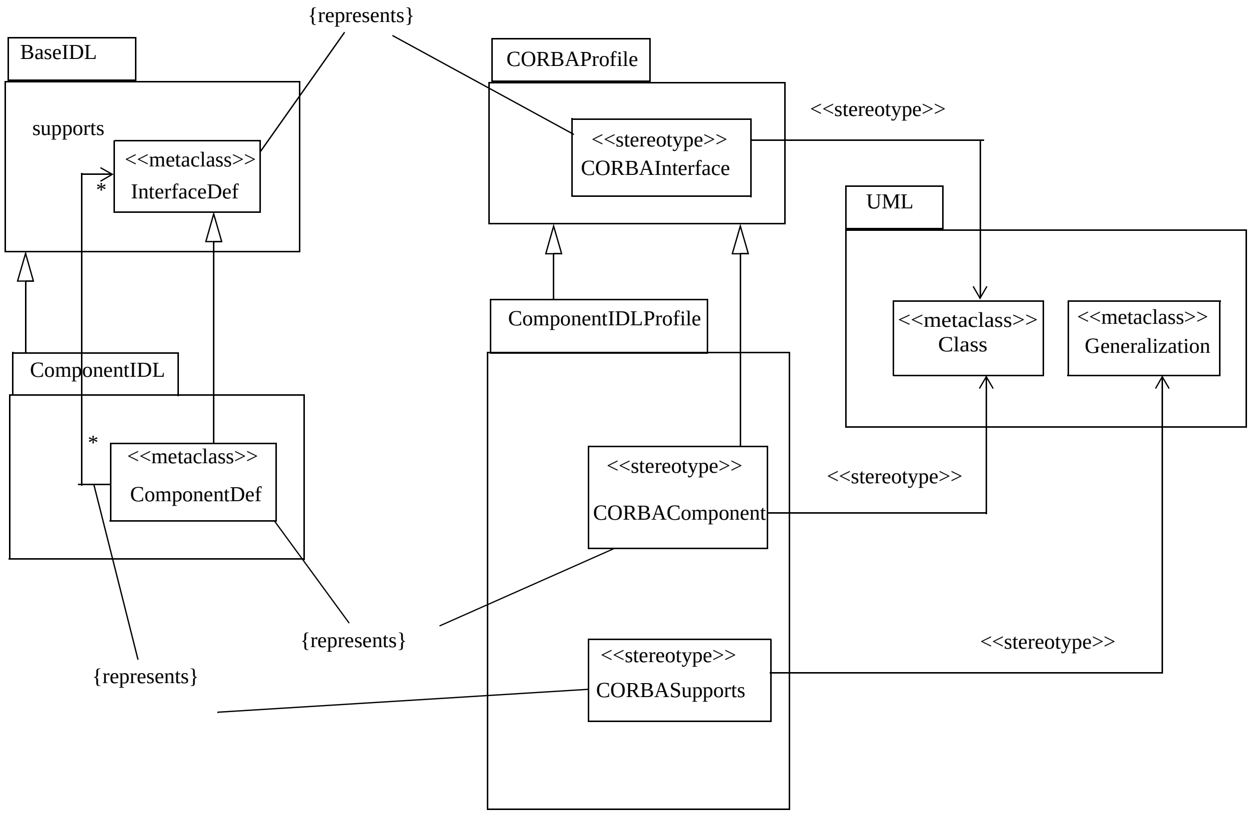

Figure 8.10 - Component mapping

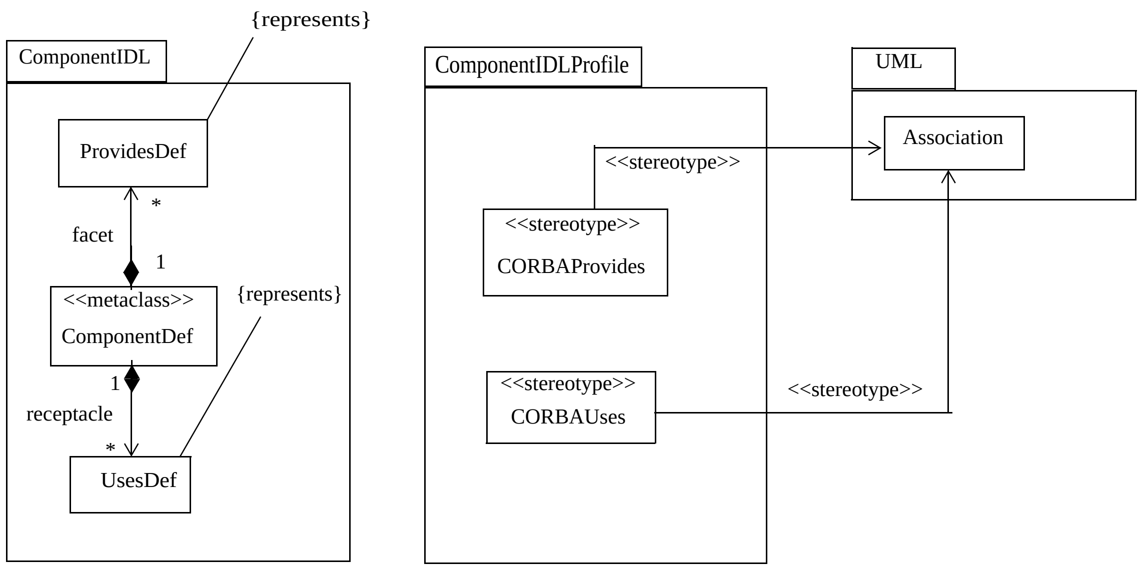

Figure 8.11 - Ports mapping

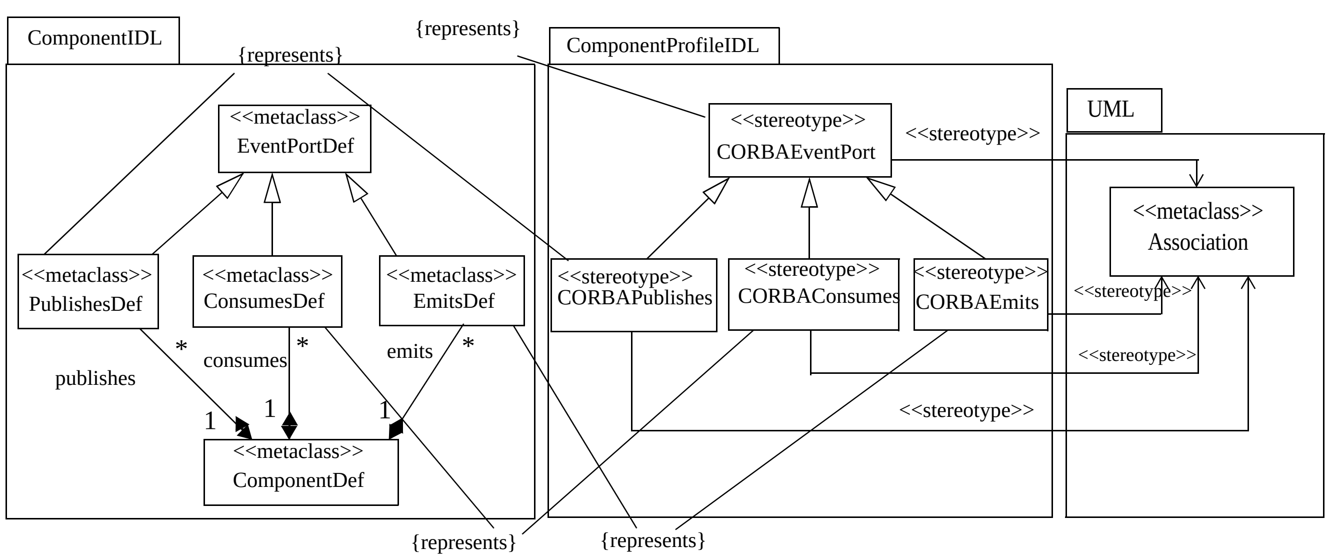

Figure 8.12 - Events mapping

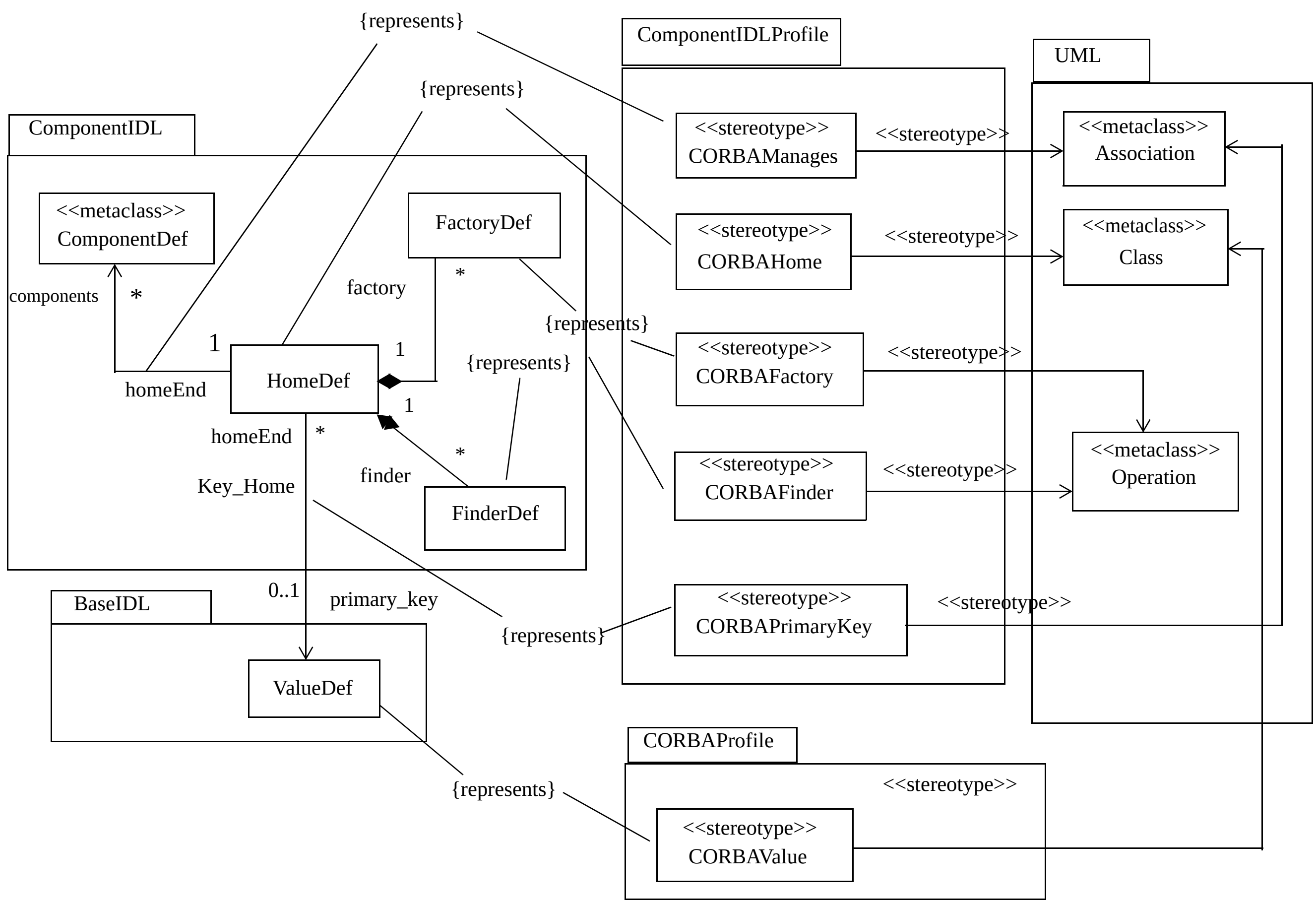

Figure 8.13 - **Home mappings**

8.2 UML Profile for CIF

8.2.1 CIF Metamodel

Figure 8.14 gives the structures of the CCM CIF concepts. A more detail description of these concepts can be found in [4].

The CIF metamodel defines additional metaclasses and associations to specify how a component has to be implemented.

The component implementation (ComponentImplDef) is used to model an implementation definition for a given component definition. It specifies an association to component definition to allow instances to point exactly to the component the instance is going to implement.

Segment type (SegmentDef) is used to model a segmented implementation structure for a component implementation. The behavior for each component feature (ComponentFeature) can be provided by a separate segment of the component implementation.

Segment type has in addition an association to an Artifact type (ArtifactDef), which is a model of programming language constructs (e.g., classes) used to actually implement the behavior for component features.

Segment definitions modeled as instances of the Segment type may contain a set of policies (Policy), which have to be applied to realizations of the segment in the implementation code. These policies include for example activation policies for the artifact associated to a segment. The complete set of required policies is not defined yet, so the metamodel is flexible in this case. In the CCM Profile Policy concept is not for interest and is not considered further.

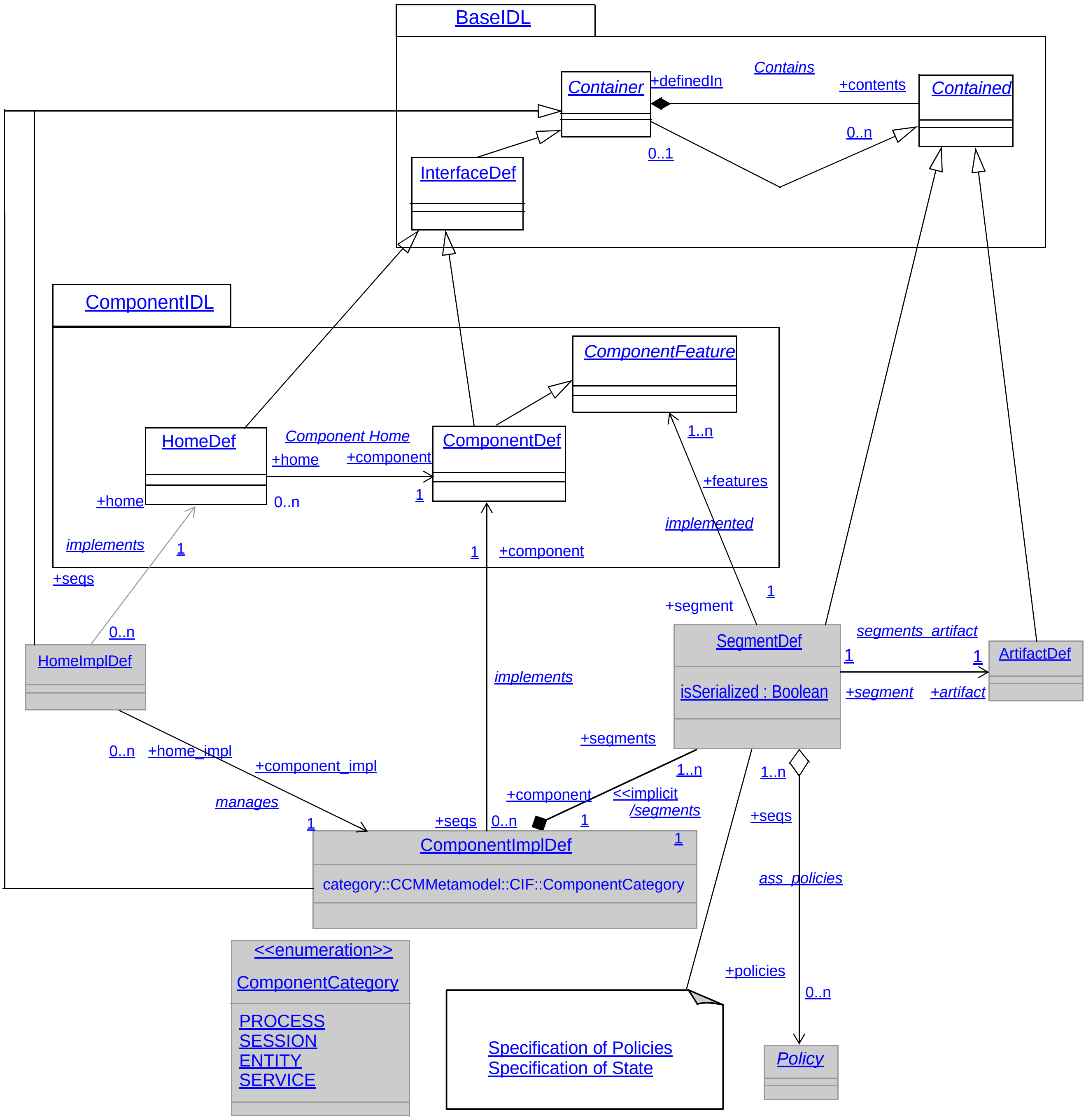

Figure 8.14 - CIF metamodel

8.2.2 Profile Definition

Component implementation

A component implementation is defined using a UML class with the stereotype “CORBAComponentImpl.”

Table 8.6 - CORBAComponentImpl and CORBAImplements stereotypes

Stereotype	Base Class	Parent	Tags	Description
CORBAComponentImpl <<CORBAComponentImpl>>	Class	NA	category	A CORBAComponentImpl is a class for implementation definition for a given component definition.
CORBAImplements <<CORBAImplements>>	Association	NA		CORBAImplements is an association between components and component implementations and between homes and home implementations.

Table 8.7 - Tag definition for CORBAComponentImpl Stereotype

Tag	Stereotype	Type	Multiplicity	Description
category	CORBAComponentImpl	ComponentCategory	1	Indicates the life cycle category of the component implementation. CCM specifies four categories of the component implementation: session, entity, process, and service.

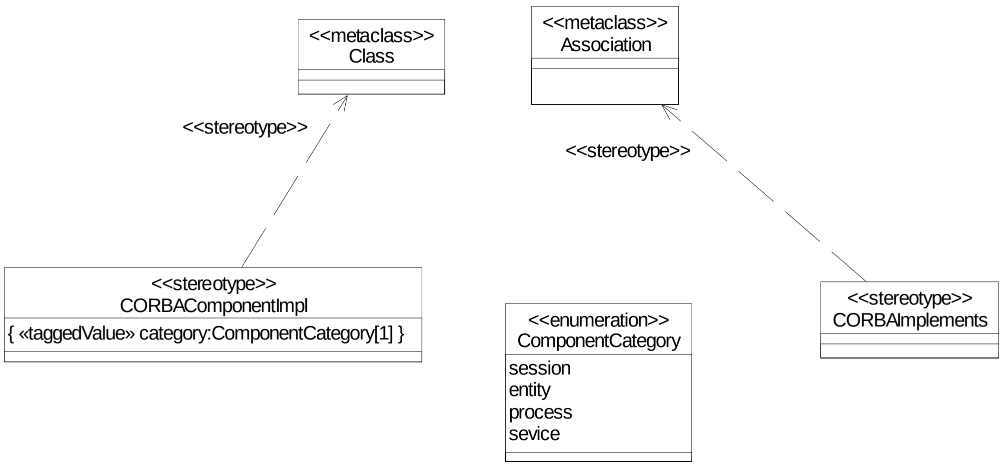

Figure 8.15 - Explicit Modeling of CORBAComponentImpl and CORBAImplements stereotypes

<<CORBAComponentImpl>> constraints

- There is an association between <<CORBAComponentImpl>> and <<CORBAComponent>>.

  self.connection→
  exists(participant.isStereotyped("CORBAComponentImpl")) and self.connection→exists(participant.isStereotyped("CORBAComponent"))

- The only classes that are allowed to be contained by a <<CORBAComponentImpl>> are classes with the stereotype <<CORBASegment>>.

  self.connection→
  exists(participant.isStereotyped("CORBAComponentImpl") and aggregation = #composite and aggregation.participant.isStereotyped("CORBASegment"))

<<CORBAImplements>> contraints

- A <<CORBAComponentImpl>> always has exactly one <<CORBAComponent>> associated while each <<CORBAComponent>> might be implemented by different types of <<CORBAComponentImpl>>.

  self.connection→
  exists(participant.isStereotyped("CORBAComponentImpl") and multiplicity.min=1 and max=*)
  self.connection→exists(participant.isStereotyped("CORBAComponent") and multiplicity.min=1 and max=1)

- Each << CORBAHomeImpl >> in a model implements exactly one <<CORBAHome>>.

  self.connection→exists(participant.isStereotyped("CORBAHomeImpl") and multiplicity.min=1 and max=1)
  self.connection→exists(participant.isStereotyped("CORBAHome") and multiplicity.min=1 and max=1)

<<CORBAManages>> constraints

- It's an association between a «CORBAHomeImpl» and a «CORBAComponentImpl».

  self.connection→
  exists(participant.isStereotyped("CORBAHomeImpl")) and self.connection→

exists(participant.isStereotyped("CORBAComponentImpl"))

- Each << CORBAHomeImpl >> manages exactly one <<CORBAComponentImpl>>, this relation is modeled by the association <<CORBAManages>>.

self.connection→
exists(participant.isStereotyped("CORBAComponentImpl") and multiplicity.min=1 and max=1)

Home implementation

A home implementation of a component is defined using a UML class with the stereotype "CORBAHomeImpl."

Table 8.8 - CORBAHomeImpl stereotype

Stereotype	Base Class	Parent	Tags	Description
CORBAHomeImpl <<CORBAHomeImpl>>	Class	NA	NA	A CORBAHomeImpl is a class for implementation definition for a given home definition.

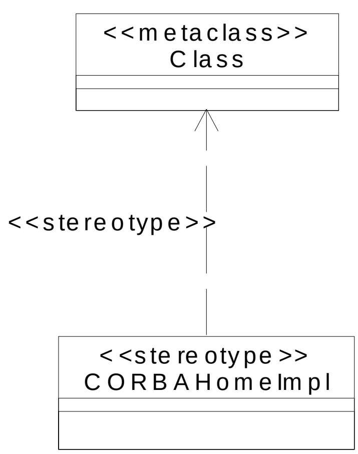

Figure 8.16 - Explicit Modeling of CORBAHomeImpl stereotype

<<CORBAHomeImpl>> contraints

- For each instance x of <<CORBAHomeImpl>> the instance of <<CORBAComponent>>, which is associated to the instance of <<CORBAHome>> associated to x is the same instance as the instance of <<CORBAComponent>> associated to the instance of <<CORBAComponentImpl>>, which is associated to x.

self.home.component = self.component_impl.component

- The life cycle category of the <<CORBAComponentImpl>> must be "***entity***" or "***process***" if the component implementation is segmented.

self.segments>1 implies (self.category=ENTITY or self.category=PROCESS)

Example

The following IDL3 example describes a representation of the minimal form as a composition (without Managed Storage), which specifies a unit of component implementation.

```
component ExmplCom {};
home ExmplHome manages ExmplCom {};
composition session ExmplComImpl {
    home executor ExmplHomeImpl{
        implements    ExmplHome;
        manages       ExmplComSessionImpl;
    };
};
```

Using the UML Profile for CCM the described composition above can be represented with the following UML model:

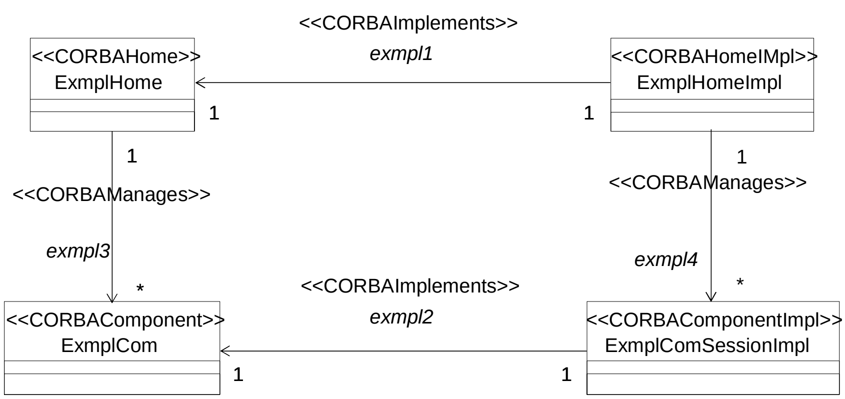

Figure 8.17 - CIDL composition: unit of a component implementation in CCM

Segment and artifact

A segment of a component implementation is defined using a UML class with the stereotype “CORBASegment.”

An artifact of a component implementation is defined using a UML class with the stereotype “CORBAArtifact.”

Table 8.9 - CORBASegment and CORBAArtifact stereotypes

Stereotype	Base Class	Parent	Tags	Description
CORBASegment <<CORBASegment>>	Class	NA	isSerialized features	A CORBASegment is a class that is used to model a segmented implementation structure for a component implementation. This means that the behavior for each component feature can be provided by a separate segment of the component implementation.
CORBAArtifact <<CORBAArtifact>>	Class	NA	NA	A CORBAArtifact is a class that represents the abstractions from programming language constructs like Classes.

Table 8.10 - Tag definition for CORBAComponentImpl Stereotype

Tag	Stereotype	Type	Multiplicity	Description
isSerialized	CORBASegment	Boolean	1	Indicates that the access to segment is required to be serialized or not.
features	CORBASegment	String	1..n	Indicates which component feature is provided by the segment.

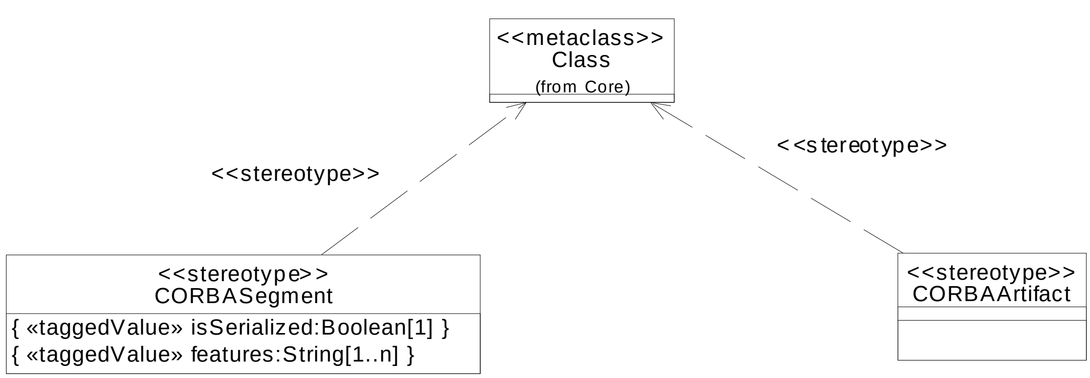

Figure 8.18 - Explicit Modeling of CORBASegment and CORBAArtifact stereotype

<<CORBASegment>> contraints

- <<CORBASegment>> classes are always contained in <<CORBAComponentImpl>>.

  self.definedIn.oclIsTypeOf(ComponentImplDef)

<<CORBAArtifact>> contraints

- The only allowed *Container* for *ArtifactDef* is *ModuleDef*.

  self.definedIn.oclIsTypeOf(ModuleDef)

Example

The following IDL3 example extends the previous example to illustrate segmented executors (component implementation). A segmented executor **ExmplComEntityImpl** is a set of physically distinct artifacts **ExmplFacet1** and **ExmplFacet2**.

```
component ExmplCom {};
home ExmplHome manages ExmplCom {};
composition entity ExmplComImpl {
    home executor ExmplHomeImpl{
        implements    ExmplHome;
        manages       ExmplComEntityImpl{
    segment ExmplSeg1{
        provides (ExmplFacet1); };
    segment ExmplSeg2{
        provides (ExmplFacet1); };
    };
  };
};
```

Using the UML Profile for CCM the described composition above can be represented with the following UML model:

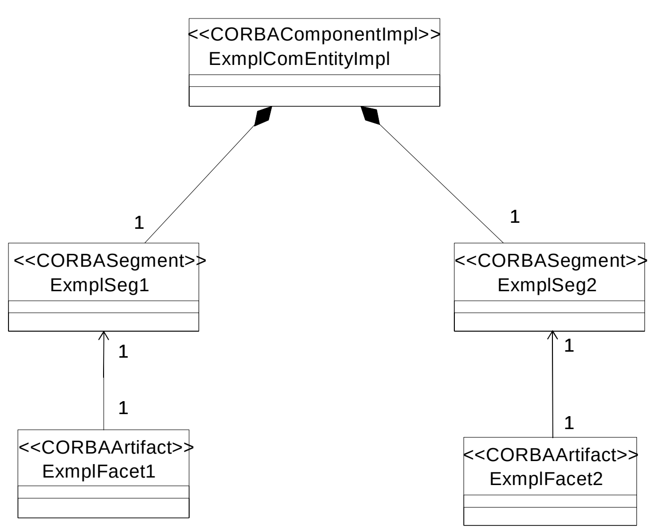

Figure 8.19 - Segments and artifacts

8.2.3 Metamodel to Profile Mapping

The mapping between the profile and the metamodel of the CCM is specified by giving the relation between the metamodel elements and the elements of the profile. It is shown in the following figure. The graphical relation "represents" means that the specific model element of the metamodel is represented by the associated construct(s) of the profile. For example, an instance of the metaclass ComponentImplDef would be represented by a UML class stereotyped as CORBAComponentImpl.

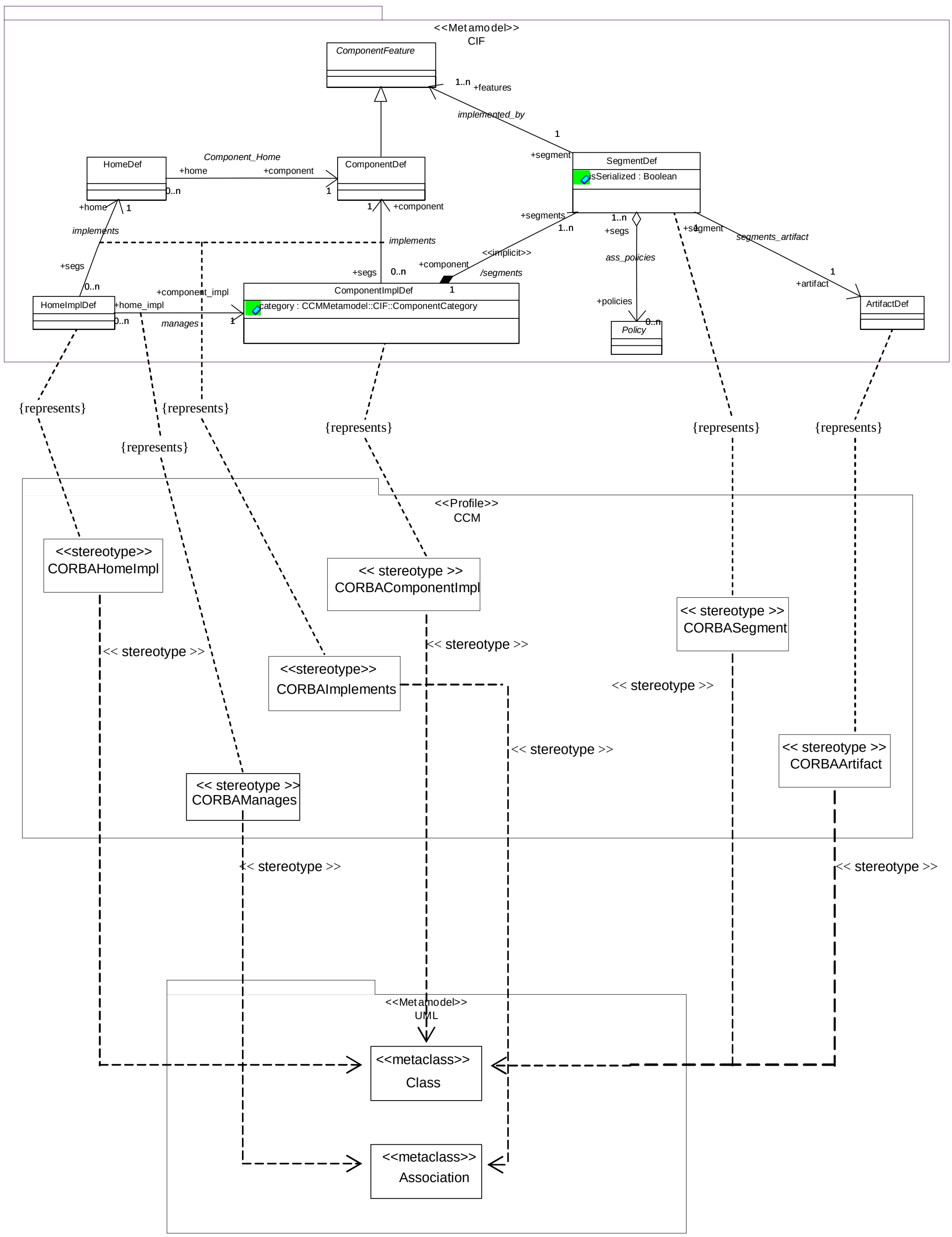

Figure 8.20 CIF metamodel to CCM Profile and CCM Profile to UML mapping

9 Profile Illustration with the Dinning Philosopher

9.1 Example Scenario Description

The example scenario includes three different types of components:

1. Philosopher
2. Fork
3. Observer

A configurable number of philosophers (active components) are sitting around a table. Philosophers perform actions: thinking, eating, and sleeping. They do not need any resources in order to think or to sleep, but they need two forks in order to eat, one for the left hand and one for the right hand. Therefore, before starting to eat, a philosopher tries to get two forks.

An observer will be notified by all philosophers in case of an activity change (when a philosopher starts eating, starts thinking, or starts sleeping). Furthermore, the critical state of getting hungry is notified to an observer as well.

9.2 Type Definition

The example uses the following IDL3 basic types and exceptions:

```
module Dinner {
exception InUse{};
Exception TooMuchPhilosopher{};

typedef string PhilosopherName;
typedef enum PhilosopherState {
    EATING,
    THINKING,
    HUNGRY,
    STARVING,
    DEAD
    };
}
```

The Type and exception definition model gives the same information using the CORBA UML profile.

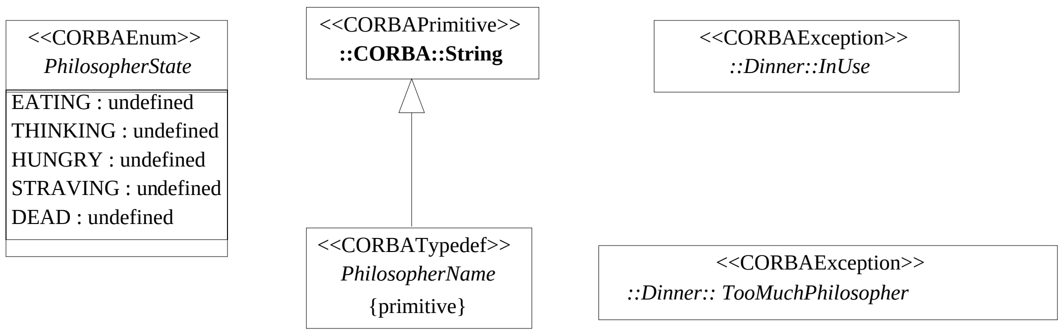

Figure 9.1 - Type and exception definition

9.3 Interface Definition

The interfaces needed for port definitions are the following IDL3 definition:

```
Module Dinner {

interface Registration {
        PhilosopherName register()raises (TooMuchPhilosopher);
    };
interface Fork {
        void get() raises (InUse);
        void release();
    };
};
```

The Interfaces definition model gives the same information using the CORBA UML profile.

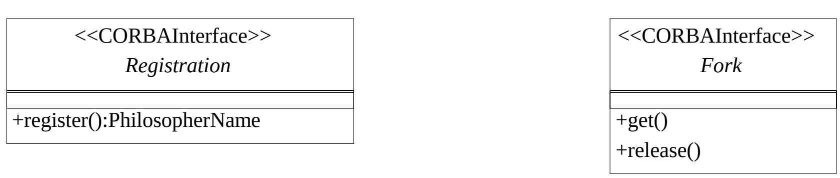

Figure 9.2 - Interfaces definition

9.4 Component Definition

The IDL3 component definitions are the following:

```
module Dinner {
eventtype StatusInfo {
    public PhilosopherName  name;
    public PhilosopherState state;
```

```
    public long secondesSinceLastMeal;
    public boolean hasLeftFork;
    public boolean hasRightFork;
};

    component Philosopher {
        readonly attribute long metabolicRate;
        uses Fork leftFork;
        uses Fork rightFork;
        uses Registration registration;
        publishes StatusInfo statusInfo;
    };

    component Fork {
        provides Fork one_fork;
    };

    component Registrator supports Registration {
    };

    component Observer {
        consumes StatusInfo info;
    };
};
```

The Philosopher external view model, the Fork component external view model, the Registrator external view model, and Observer external view model give the same information using the CCM UML profile.

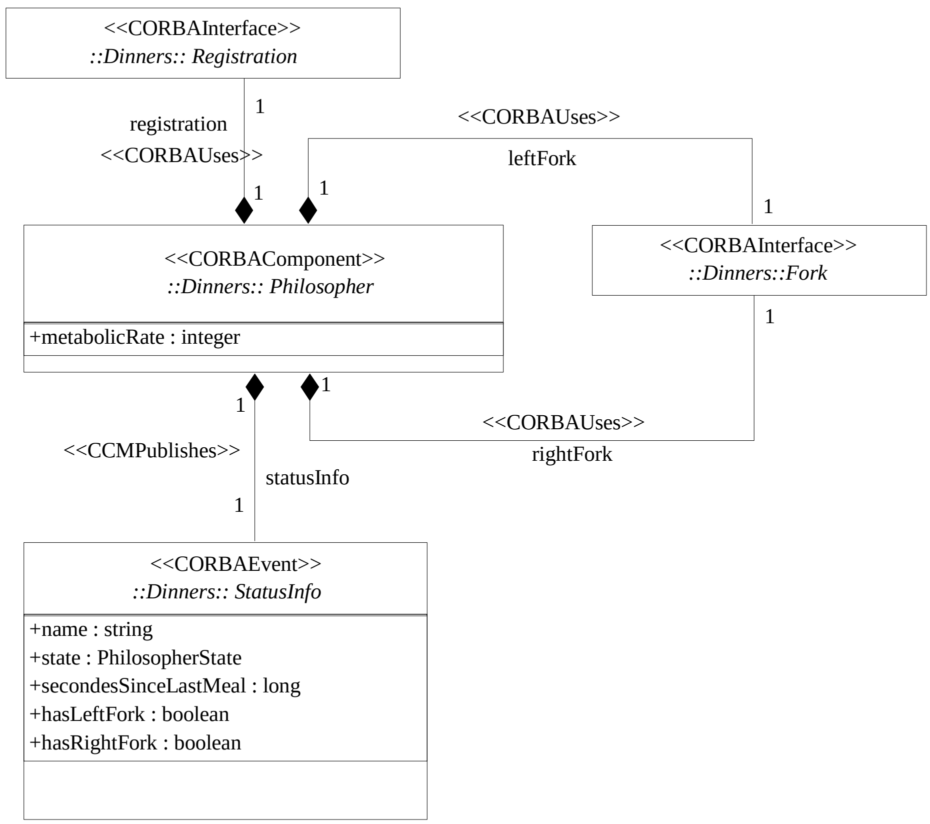

Figure 9.3 - Philosopher external view

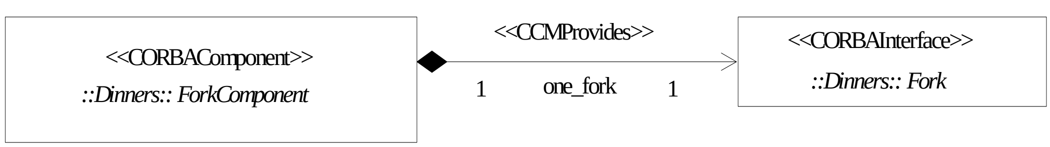

Figure 9.4 - ForkComponent external view

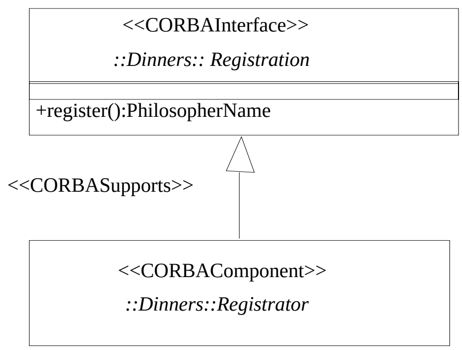

Figure 9.5 - Registrator external view

Figure 9.6 - Observer external view

9.5 Home Definition

The IDL3 home definitions are the following:

module Dinner {
home RegistratorHome manages Registrator {};
home PhilosopherHome manages Philosopher {};
home ForkHome manages Fork {};
home ObserverHome manages Observer {};
};

The Registrator home, Philosopher home, Fork home and Observer home give the same information using the CCM UML profile.

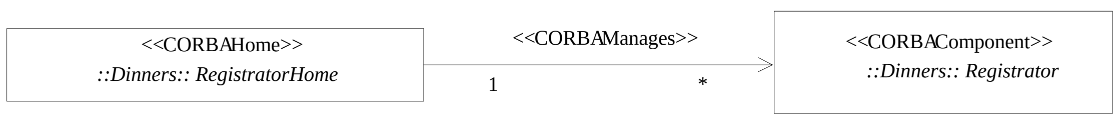

Figure 9.7 - Registrator home

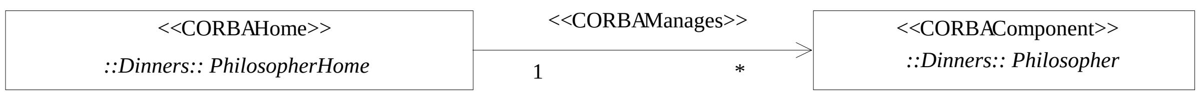

Figure 9.8 - Philosopher home

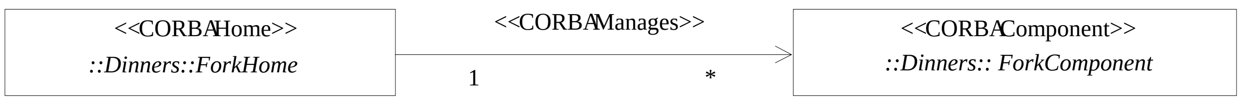

Figure 9.9 - Fork home

Figure 9.10 - Observer home

9.6 Component Implementation Definition

The IDL3 component implementation definition is a composition. For each component the following composition descriptions were defined:

```
module Dinner {

composition session PhilosopherImpl {
home executor PhilosopherHomeImpl {
implements PhilosopherHome;
manages PhilosopherSessionImpl;
};
};

composition entity ForkImpl {
home executor ForkHomeImpl {
implements ForkHome;
manages ForkEntityImpl {
segment Seg { provides the_fork; }
```

```
};
};
};

composition session ObserverImpl {
home executor ObserverHomeImpl {
implements ObserverHome;
manages ObserverSessionImpl;
};
};

composition session RegistratorImpl {
home executor RegistratorHomeImpl {
implements RegistratorHome;
manages RegistratorSessionImpl;
};

};
};
```

The implementations for the components: Fork, Philosopher, Registrator, and Observer in form of compositions described above have the following representations using the UML Profile for CCM:

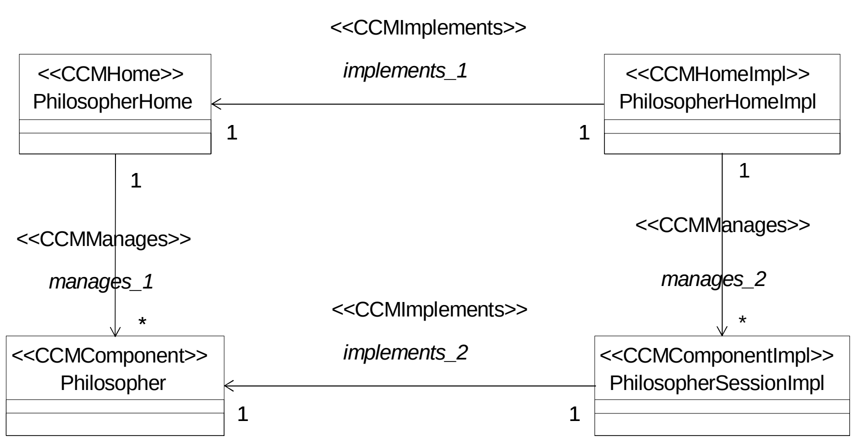

Figure 9.11 - The Composition model for the Philosopher component

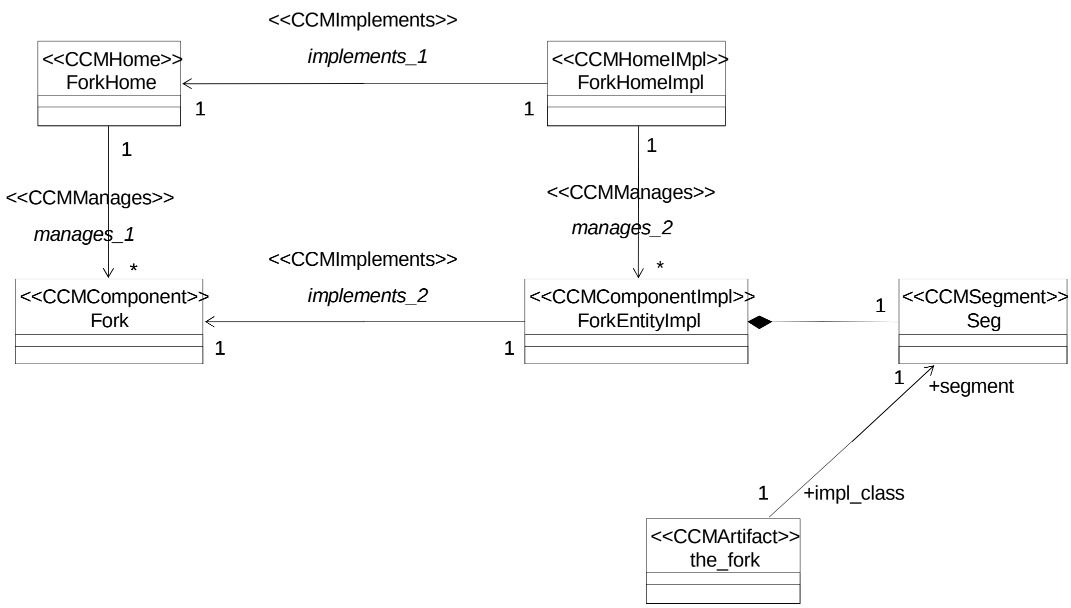

Figure 9.12 - The Composition model for the ForkComponent

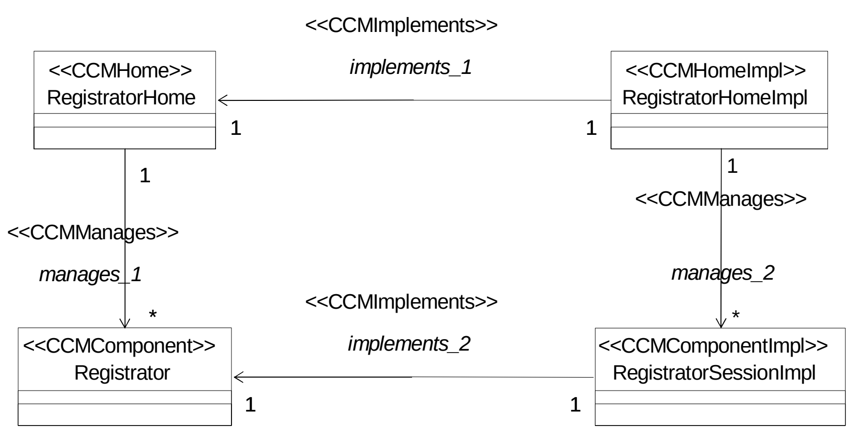

Figure 9.13 - The Composition model for the Registrator component

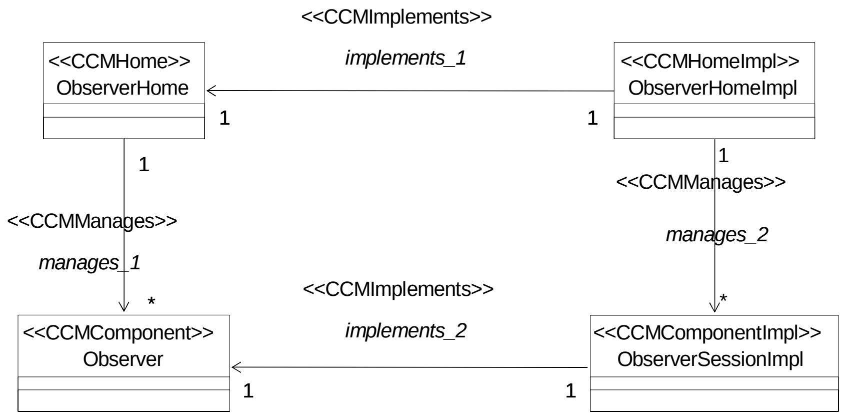

Figure 9.14 - The Composition model for the Observer component

Annex A
(normative)

References

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

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

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

[4] CORBA Components Specification, OMG TC Document formal/02-06-65

[5] www.puml.org/mml

[6] Unified Modeling Language (UML) Specification, Version 1.5, OMG TC Document formal/03-03-01

[7] The UML Profile for CORBA, OMG TC Document formal/02-04-01

[8] Virtual metamodel for the UML Profile for CCM (DTD), OMG document: ptc/2005-01-01

[9] Virtual metamodel for the UML Profile for CCM (MDL), OMG document: ptc/2005-01-02

[10] Virtual metamodel for the UML Profile for CCM (XML), OMG document: ptc/2005-01-03

INDEX