Date: April 2008

UML Profile for CORBA and CORBA Components Specification

Version 1.0

OMG Document Number: formal/2008-04-07
Standard document URL: http://www.omg.org/spec/CCCMP/1.0/PDF
Associated files*: http://www.omg.org/spec/CCCMP/20070501 (XMI)
http://www.omg.org/spec/CCCMP/20070502 (Metamodel)

* original files: ptc/07-05-15 (XMI), ptc/07-05-16 (Metamodel)

OMG's Issue Reporting Procedure

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

Table of Contents

Preface

About the Object Management Group

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. A catalog of all OMG Specifications is available from the OMG website at:

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

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

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

Platform Specific Model and Interface Specifications

- CORBAserviceCORBAfacilities

- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. (as of January 16, 2006) at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: pubs@omg.org

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

Intended Audience

This specification is intended primarily for CORBA and CORBA Component Model (CCM) vendors and CORBA/CCM tools developers. End-users may find the specification useful to design CORBA and/or CCM based applications.

Typographical Conventions

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

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

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

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

Helvetica/Arial - 10 pt: Exceptions

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

Issues

Readers are encouraged to report any technical or editing issues/problems with this specification 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*.

1 Scope

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's standard architecture for distributed object systems. CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB).

CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

The CORBA Component Model (CCM) is a comprehensive component standard based on the reliable and well-proven CORBA architecture. 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/06-04-01 and the existing UML Profile for CORBA Components specifies how to represent these concepts using UML 1.5. The new version of UML (UML2.1) has brought new powerful concepts like Structured Classifiers "Port" or "Part," and improved the existing concepts like "Component" and "Interface."

This specification provides a UML2 profile that facilitates representation of concepts needed to represent a pure CORBA or CORBA Components PSM. In conjunction with existing OMG specifications, namely UML2, CORBA, CORBA Components, and the MOF2, this will result in significant benefits to the CORBA and CORBA Components user community and the users of MDA in general.

2 Conformance

This specification defines three mandatory conformance points. All CCM Profile implementations must support these conformance points:

- Implementation of the UML Profile for CORBA, defined in Section 8.1.
- Implementation of the ComponentIDL Profile, defined in Section 8.2.
- Implementation of the CIF Profile, defined in Section 8.3.

An implementation of the Deployment Profile defined in Section 8.4 and CCMQoS Profile defined in Section 8.5 is optional. Nevertheless, it is recommended to provide deployment and QoS support for CCM Profile implementation.

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.

- CORBA/IIOP Specification, Version 3.0.3
- MOF 2.0 Core Specification
- UML 2.1 Infrastructure Specification

- UML 2.1 Superstructure Specification
- UML 2.0 OCL Specification
- CORBA Component Model Specification, Version 4.0
- Streams for CCM Specification
- Deployment and Configuration of Component-based Distributed Applications Specification
- QoS for CCM final adopted specification

4 Terms and Definitions

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

artifact

An element that describes 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
D&C	Deployment and Configuration
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

NOTE: This document shall replace the UML Profile for CORBA (formal/02-04-01) and the UML Profile for CORBA Components (formal/05-07-06).

6.2 How to Read this Specification

The rest of this document contains the technical content of this specification. As background for this specification, readers are encouraged to first read the CORBA Component Model (CCM) Specification (formal/06-04-01). This document is fully based on the concepts defined in the CCM Specification, these concepts are specified in form of MOF compliant CCM metamodel in Chapter 7. Chapter 8 provides the normative definition of the UML Profile for CORBA and CORBA Components. Chapter 9 provides the “ATM Simulation” example expressed in terms of the defined in Chapter 8 profile.

Although the chapters are organized in a logical manner and can be read sequentially, this is a reference specification and is intended to be read in a non-sequential manner. Consequently, extensive cross-references are provided to facilitate browsing and search.

6.3 Acknowledgements

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

- Fraunhofer Institute FOKUS
- Thales

NOTE: The technology described by this specification is based on the work of the Modelware project (http://www.modelware-ist.org) and the AD4 project (http://www.ad4-project.com) of the European Commission. The authors would like to thank the participants of these projects for their contributions and review activities.

7 CCM Metamodel

The CCM metamodel defines the abstract language of a modeling language that supports modeling general CCM concepts. This metamodel defines a set of modeling elements represented as metaclasses. A concrete syntax must define the specific notation rules for the graphical representation of this modeling language. In our case the modeling language is UML2 and the concrete syntax for modeling CCM applications with UML2 does not exist yet. The UML profile that will be introduced in the next section supports the representations of CCM concepts in terms of UML2 models.

7.1 Overview

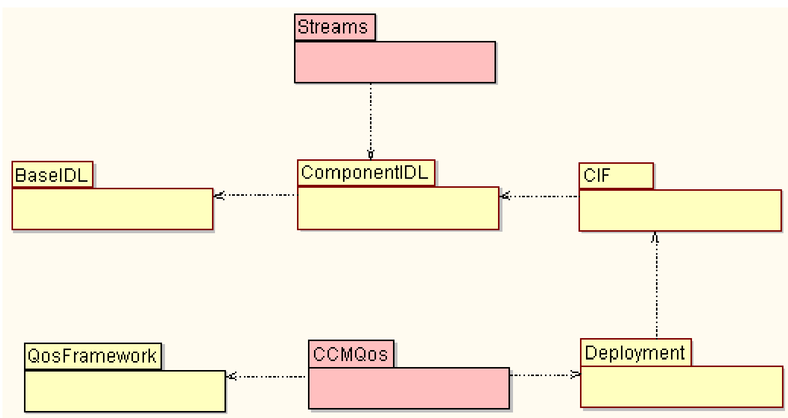

Figure 7.1- CCM Metamodel package structure

As shown in Figure 7.1 the complete CCM concept space consists of further packages: BaseIDL, ComponentIDL, CIF (Component Implementation Framework), Deployment, Streams, and CCMQoS. The QoSFramework package provides metamodel for the description of QoS properties and is defined in the “UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms” specification.

The BaseIDL package is a MOF-compliant description of the pre-existing CORBA Interface Repository. This metamodel has been standardized in formal/06-04-01.

The ComponentIDL package expresses the CORBA Component Model and based on the concepts already specified in the BaseIDL Package. This metamodel has been standardized in formal/06-04-01.

The CIF package contains metaclasses and associations for definition the programming model for constructing component implementations, and is based on the reference ComponentIDL metamodel. This metamodel has been standardized in formal/06-04-01.

The Deployment package is a MOF-compliant extended description of the Deployment and Configuration concepts for CCM. This metamodel describes concepts like assembly or component instance, and can be used for generation of XML deployment description.

The Streams package extends the CCM metamodel by providing additional means for modeling of communications of continuous data streams between CORBA components.

The CCMQoS package based on the standardized QoSFramework package mentioned above and extends the scope of the CCM metamodel to QoS property definition for CORBA components.

7.2 BaseIDL Metamodel

The first goal of the CCM metamodel is to express the extensions to CORBA IDL defined by the CORBA Component Model Standard. Since these extensions are based on the previously-existing IDL, it is not possible to define a MOF-compliant metamodel for the extensions without defining a MOF-compliant metamodel for the IDL base. Thus, the CCM Standard defined the first MOF Package, entitled BaseIDL. BaseIDL is a MOF-compliant metamodel of the pre-existing CORBA Interface Repository (IR) and contains all CORBA types. As shown by Figure 7.1, all further packages are dependent upon the BaseIDL Package.

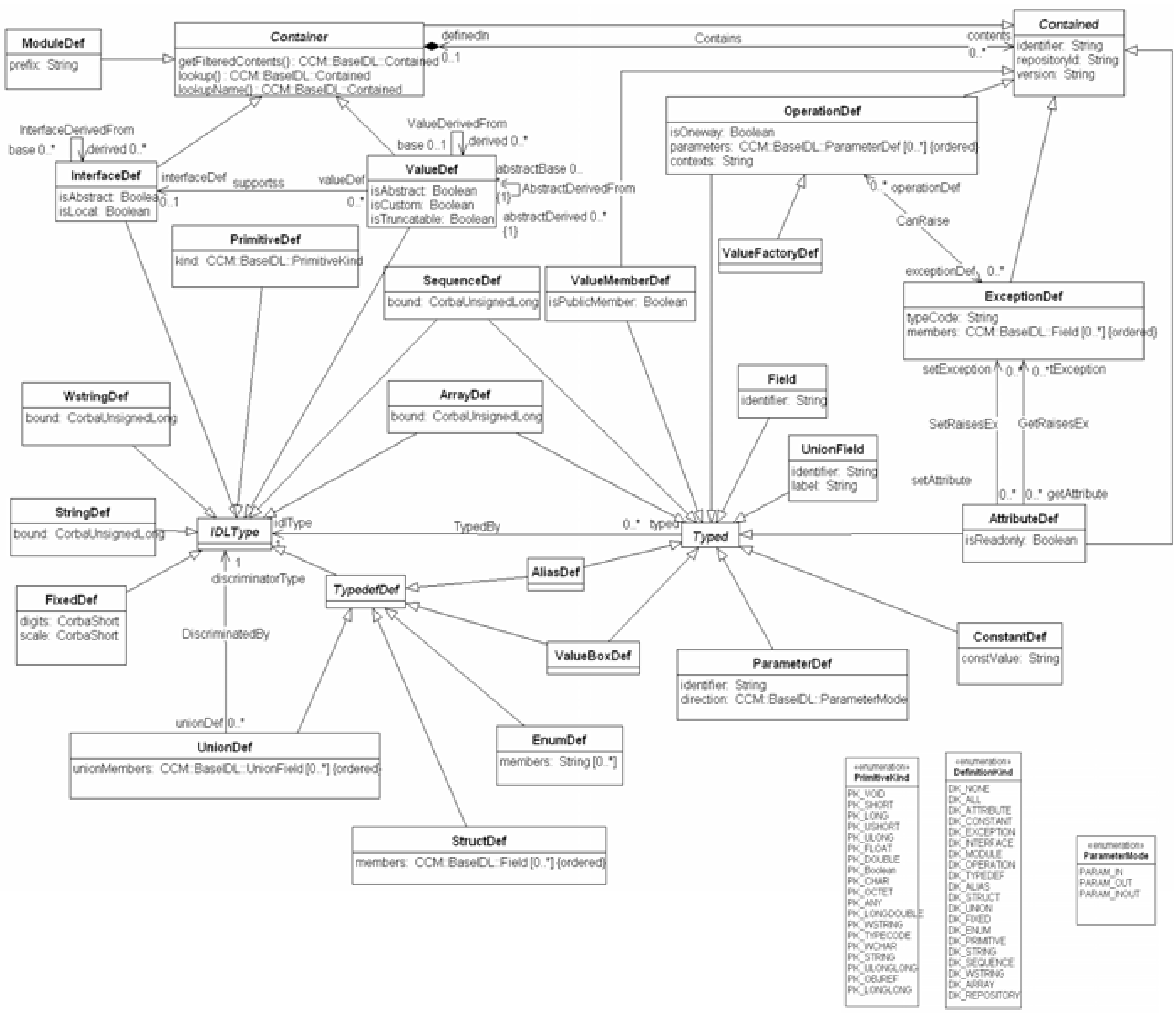

Figure 7.2 - BaseIDL Metamodel

BaseIDL definitions focus on interfaces, the operations supported by those interfaces, and exceptions that may be raised by operations. This requires quite a bit more: a large part of BaseIDL is concerned with the definition of data types. This is because data can be exchanged between client and server only if their types are defined in BaseIDL. Figure 7.2 shows all of the metaclasses and relationships defined in the BaseIDL Package.

7.2.1 Typing

In the existing CORBA IR, elements, that are "typed" such as constants, attributes, operations, etc. contain an attribute of type *IDLType*. However, the same *IDLType* can be the type for many elements, so an attribute (with its composition semantics) is not appropriate. Instead, the abstract *Typed* metaclass and an association between Typed and the *IDLType* metaclasses were specified and eliminate the need for repeating the type attribute.

The abstract metaclass *IDLType* represents OMG IDL types such as Interface, Array, or IDL primitive types such as long or string.

IDL provides a number of built-in basic types, and they are shown in Figure 7.2 by the metaclass *PrimitiveDef*. This metaclass has an attribute "*kind*" from type *PrimitiveKind*. The *PrimitiveKind* metaclass provides all inherent CORBA types like short, long, or string.

In addition to providing the built-in basic types, IDL permits you to define complex types: enumerations (*EnumDef*), structures (*StructDef*), unions (*UnionDef*), sequences (*SequenceDef*), and arrays (*ArrayDef*). You can also use typedef (*TypedefDef*) to explicitly name a type.

BaseIDL permits the definition of constants by the metaclass *ConstantDef*. This metaclass has an attribute constValue for fixed value of the constant.

For more information about CORBA types please refer to the CORBA Specification (http://www.omg.org/spec/CORBA/).

7.2.2 Containment

Many elements in the metamodel descend from Container or Contained metaclasses.

The abstract metaclass Contained is inherited by all elements that are contained by other BaseIDL elements. All elements within the BaseIDL, except definitions of anonymous (ArrayDef, StringDef, WstringDef, FixedDef, and SequenceDef), and primitive types are contained by other elements. All metaclasses derived from the Contained metaclass hold an identifier (attribute "*identifier*"), repositoryID (attribute "*repositoryID*"), and version (attribute "*version*").

The abstract metaclass *Container* is used to describe a containment hierarchy in the BaseIDL metamodel. A *Container* can contain any number of elements derived from the *Contained* metaclass. All metaclasses derived from Container are also derived from Contained.

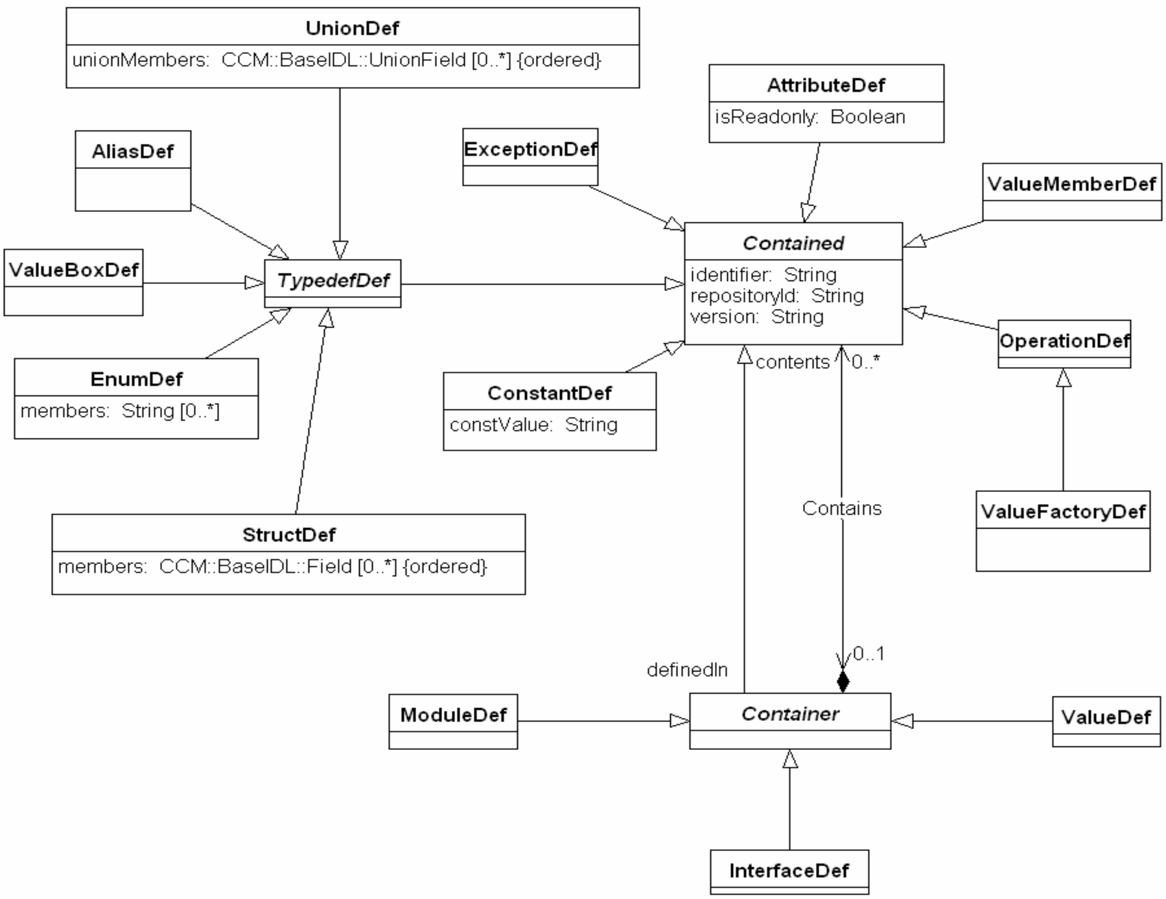

Figure 7.3 - BaseIDL Containment hierarchy

7.2.3 Modules

The metaclass *ModuleDef* defines an IDL module. IDL uses the module construct to create namespaces, therefore the *ModuleDef* metaclass is also a Container: modules combine related definitions into a logical group and prevent pollution of the global namespace. Modules can contain any definition that can appear at global scope (type, constant, exception, and interface definitions). In addition, modules can contain other modules, so nested hierarchies are also possible.

7.2.4 Interfaces

The most important metaclass in the BaseIDL is the *InterfaceDef*, which describes an IDL Interface defined as a set of operations that an instance of that interface must support. *InterfaceDef* forms a namespace and is a *Container.* You can nest the following contained elements inside an interface definition:

- *ConstDef* (Constant definitions)
- *TypedefDef* (all named non-object.type definitions like structure, union, or enumeration)
- *ExceptionDef* (Exception definitions)

- *AttributeDef* (Attribute definitions)
- *OperationDef* (Operation definitions)

InterfaceDef does not have a private or protected part. By definition, everything in an *InterfaceDef* is public. Interfaces can inherit from one or more other Interfaces (association *InterfaceDerivedFrom*).

Interfaces may be abstract (attribute "*isAbstract*") or local (attribute "*isLocal*").

7.2.5 Operations

An IDL operation is defined using the metaclass *OperationDef* and consists of:

- The type of the operation's return result (*OperationDef* inherits from *Typed* metaclass); the type may be any type that can be defined in BaseIDL. Operations that do not return a result specify the void type.
- A parameter list (attribute "*parameters*") that specifies zero or more parameter declarations for the operation.
- An optional raises expression (metaclass "*ExceptionDef*") that indicates which exceptions may be raised as a result of an invocation of this operation.
- An optional context expression (attribute "*context*") that indicates which elements of the request context may be consulted by the method that implements the operation.

The attribute "*isOneway*" specifies which invocation semantics the communication service must provide for invocations of a particular operation.

7.2.6 Attributes

The metaclass *AttributeDef* describes an IDL attribute. An attribute definition is logically equivalent to declaring a pair of accessory functions; one to retrieve the value of the attribute ("*get*"-function) and one to set the value of the attribute ("*set*"-function).

The attribute "*isReadonly*" indicates that only a "*get*"-function (the retrieve value function) is allowed.

7.2.7 Values

The metaclass *ValueDef* describes a CORBA value type. Value types share many of the characteristics of *InterfaceDef* and *StructDef* metaclasses:

- They support description of complex state (i.e., arbitrary graphs, with recursion and cycles).
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call).
- They support both public and private (to the implementation) data members.
- They support single inheritance (of valuetype: association "*ValueDerivedFrom*") and can support a single non-abstract interface (association "*supportss*").
- They also may be abstract (attribute "*isAbstract*"), custom (attribute "*isCustom*"), or truncated (attribute "*isTruncatable*").

7.2.8 Exceptions

The metaclass *ExceptionDef* permits the declaration of data type like structures, which may be returned to indicate that an exceptional condition has occurred during the performance of a request. Each IDL exception is characterized by the type of the associated return value (as specified by the attribute "*members*" in its declaration).

7.2.9 Parameters

The metaclass *ParameterDef* defines an IDL parameter contained in the IDL operation. A parameter declaration has a directional attribute "*direction*" that informs the communication service in both the client and the server of the direction in which the parameter is to be passed.

7.2.10 BaseIDL Constraints

[1] An AttributeDef can be defined within an InterfaceDef or within a ValueDef

[1] context AttributeDef inv:
self.definedIn.oclIsKindOf (InterfaceDef) or self.definedIn.oclIsKindOf (ValueDef)

[2] An OperationDef must be defined within an InterfaceDef or within a ValueDef

[2] context OperationDef inv:
self. definedIn.oclIsKindOf (InterfaceDef) or self.definedIn.oclIsKindOf (ValueDef)

[3] A ValueMemberDef must be defined within a ValueDef

[3] context ValueMemberDef inv:
self.definedIn.oclIsTypeof (ValueDef)

[4] Abstract ValueDefs may only derive from other abstract ValueDefs

[4]context ValueDef inv:
self.isAbstract implies base->isEmpty

[5] base element (if any) refers to a concrete ValueDef

[5] context ValueDef inv:
self.base->notEmpty implies not self.base.isAbstract

[6] AbstractBase refers only to abstract ValueDef metaclass instances

[6] context ValueDef inv:
self.abstractBase->forAll(self.isAbstract)

[7] Abstract InterfaceDefs may only derive from other abstract InterfaceDef metaclass instances

[7] context InterfaceDef inv:
self.isAbstract implies base->forAll (isAbstract)

[8] Contained elements have unique names within their Container

[8] contextContained inv:
contents->forAll (c0, c1 | c0 <> c1 implies c0.identifier <> c1.identifier)

7.3 ComponentIDL and Streams Metamodels

The following UML class diagram describes a metamodel representing the extensions to CORBA IDL defined by the CORBA Component Model. These extensions are dependent on the types defined in the CORBA Core and called ComponentIDL, so the metamodel ComponentIDL is dependent on the metamodel BaseIDL representing the base IDL and introduced in the previous section.

All ComponentIDL concepts depicted in the ComponentIDL metamodel are described in the CORBA Components Specification, thus, for more details please refer to the document formal/06-04-01.

Figure 7.4 shows the extended metaclasses from the BaseIDL metamodel *InterfaceDef*, *ValueDef*, and *OperationDef* indicated with grey color.

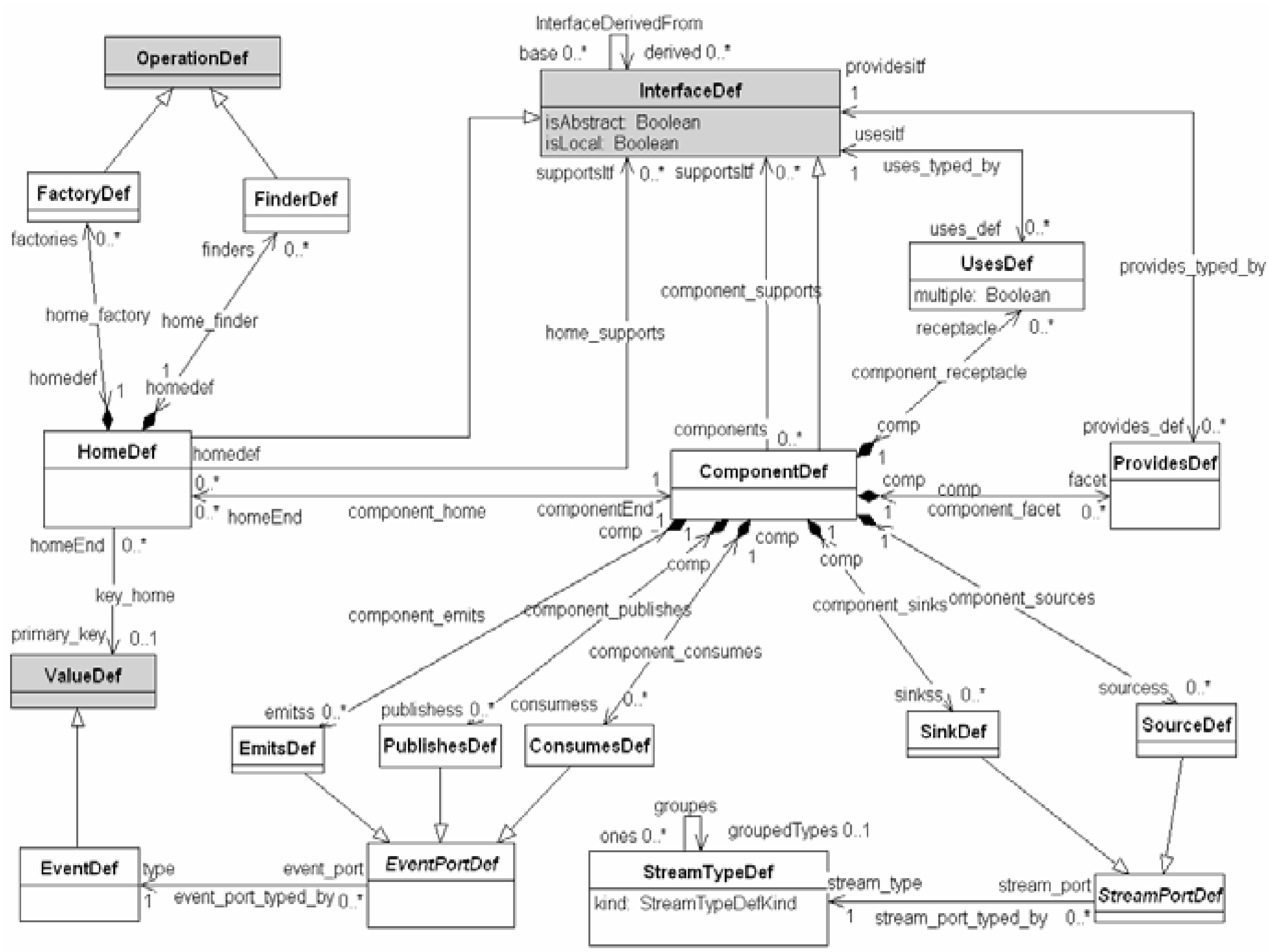

Figure 7.4 - ComponentIDL Metamodel

7.3.1 Component Model

The central metaclass in the ComponentIDL metamodel (Figure 7.4) is the *ComponentDef* metaclass that represents a CORBA Component type. A component definition in the CORBA Components Specification implicitly defines an interface (*InterfaceDef*) that supports (see the association "*component_supports*" between *ComponentDef* and *InterfaceDef* metaclasses) the features defined in the component definition body. ComponentDef metaclass extends the concept of an interface definition (inherits from *InterfaceDef* metaclass) to support features that are not supported in interfaces. Component definitions also differ from interface definitions in that they support only single inheritance from other component types but they can inherit from more than one interface (the association "*component_supports*").

Components support a variety of surface features through which they can interact with each other. These surface features are called ports. The *ComponentDef* metaclass supports four kinds of ports (facets, receptacles, event sink, and event source) defined in the CCM Standard and two additional kinds of ports - stream sink and stream source defined in the Streams for CCM Specification (ptc/05-07-01):

1. The metaclass *ProvidesDef* represents facets, which are interfaces (*InterfaceDef*) provided by the component. It is a synchronous operational communication mechanism between components.
2. The metaclass *UsesDef* represents receptacles, which are named ports that define the component's ability to use a reference supplied by other components. There are two receptacle kinds: simplex receptacles can only use a single reference, multiplex receptacles can use several references. The boolean attribute "*multipleItf*" represents the kind of receptacles. It is a synchronous operational communication mechanism between components.
3. The metaclass *ConsumesDef* represents event sinks, which are named ports into which events of a specified type may be pushed. It is an asynchronous communication mechanism between components.
4. The metaclasses *EmitsDef* and *PublishesDef* represent event sources, which are named ports that emit events of a specified type to one (*EmitsDef*) or more (*PublishesDef*) interested event consumers. It is an asynchronous communication mechanism between components.
5. The metaclass *SinkDef* represents stream sinks, which are named ports into which continuous data called streams of a specified type may be pushed. It is an asynchronous communication mechanism between components.
6. The metaclass *SourceDef* represents stream sources, which are named ports that emit continuous data of a specified type to the stream consumer. It is an asynchronous communication mechanism between components.

As described above CORBA component model supports a publish/subscribe event model and contains event type declaration (metaclass *EventDef*), which is a restricted form of CORBA value type (inherits from the BaseIDL metaclass *ValueDef*). The metaclass *EventPortDef* is an abstract class for all event ports.

7.3.2 Component Homes

CORBA Components are managed by homes (metaclass *HomeDef*, see Figure 7.4), which are CORBA Interfaces (inherit from *InterfaceDef*) providing operations to manage component life cycles, and optionally, to manage associations between component instances and primary key values (association *key_home*). Components are independent of their homes; however, a home must specify exactly one component that it manages (see the multiplicities of the association *component_home*). Multiple different home types can manage the same component type, though they cannot manage the same set of component instances.

A home may include zero or more operation declarations, where the operation may be a factory operation (*FactoryDef*), a finder operation (*FinderDef*), or a normal operation or attribute.

7.3.3 Streams

As mentioned above, the CORBA Component Model supports two different kinds of communication: synchronous operational communication and asynchronous event communication. The OMG document “Streams for CCM Specification” (ptc/05-07-01) extends the CORBA Component Model with an optional conformance point: native support for the communication of continuous data streams between CORBA components. It extends the BaseIDL and ComponentIDL metamodels standardized in CORBA Components Specification with constructs to model stream-specific ports on a component shown in Figure 7.4. Furthermore, it defines a stream type to be used for the classification of data streams:

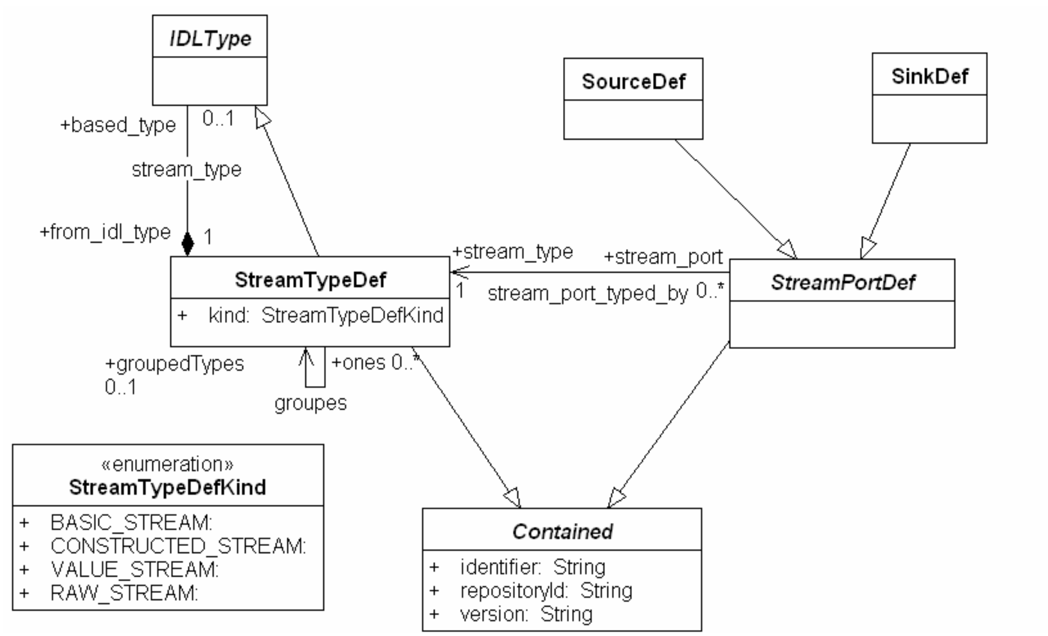

Figure 7.5 - CCM Stream Metamodel

Figure 7.5 introduces an additional kind of port for the communication of stream data, called a stream port (abstract metaclass *StreamPortDef*). A stream port can be a source port or a sink port (see Figure 7.4). A source port produces stream data of a stream type. A sink port consumes stream data of a stream type. The metaclass *StreamTypeDef* represents a stream type. The stream type may be of kind (attribute “kind” of the metaclass *StreamTypeDef*) basic, constructed, value, or raw defined by *StreamTypeDefKind* enumeration (from ptc/2005-07-01):

- Basic stream types are defined as concrete data formats for the information content of streams, which are not necessarily defined using IDL, but are encoded some other way. Basic streams are used to transport streams of encoded data, typically audio or video data. The data is consumed and produced by component implementation logic as octet sequences.
- Value stream types are a subtype of basic streams, which transport consecutive marshaled instances of data types specified in a subset of IDL. If the IDL data type of a value stream is octet, it is indistinguishable from a basic stream that does not have a specified IDL data type, and is not considered a value stream type.
- Constructed stream types are a hierarchical grouping of multiple basic stream types or other constructed types, indicating the ability to produce or consume any of the basic or value types.

- Raw streams are not typed, and intended for applications where the format of the stream does not influence the functionality of a component. Examples for the applicability of this type are a component that encrypts or compresses a data stream or a component that reads from or writes to a file.

7.3.4 Containment

The following UML class diagram describes the derivation of the metamodel elements from the BaseIDL Container and Contained elements:

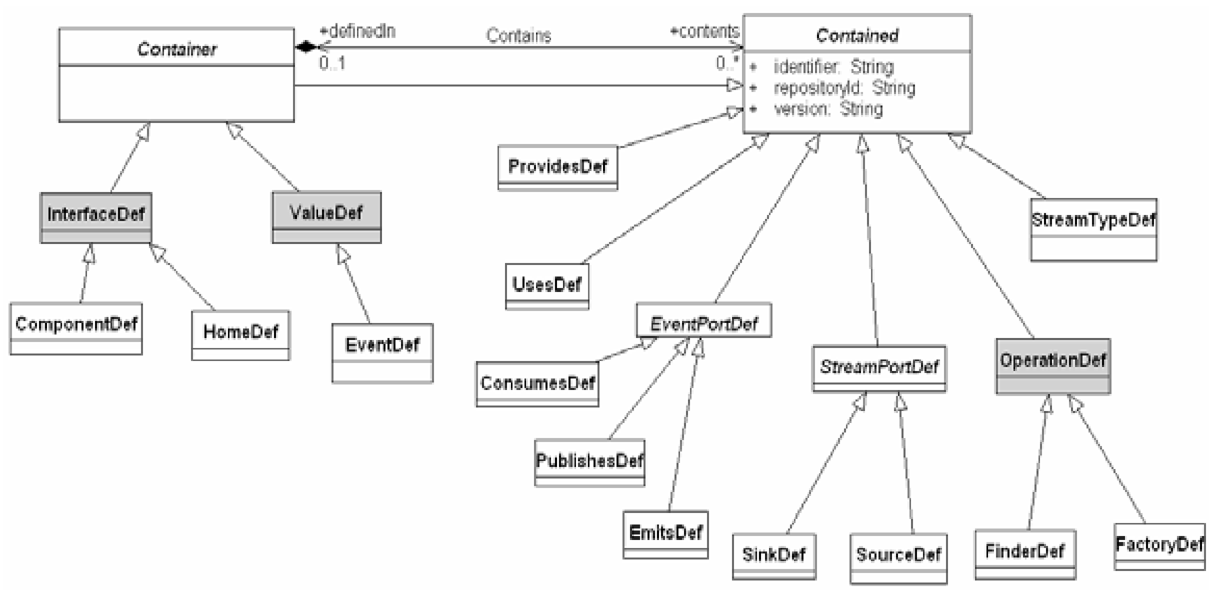

Figure 7.6 - ComponentIDL Containment hierarchy

Since the *ComponentDef* and *HomeDef* metaclasses inherit from *InterfaceDef* they form also naming scopes.

7.3.5 ComponentFeature

An instance of *ComponentDef* describes a CORBA Component in an abstract manner. The definition contains a description of all features of a component that are visible from the outside. In detail, the features supported by a CORBA Component are:

- The component equivalent interface, containing all implicit operations, operations and attributes that are inherited by a component (also from supported interfaces), and attributes defined inside the component.
- The facets of a component; that is, all interfaces that are provided by the component to the outside.
- The receptacles of a component; that is, all interfaces that are used by a component.
- The events, which a component can emit, publish, or consume.
- The streams, which a component can produce or consume.

If a component is going to be implemented, all these features must be handled by the component implementation. To provide a common basis for defining the related implementation definitions (as part of CIF) the abstract metaclass ComponentFeature is defined. The metaclasses *ComponentDef*, *ProvidesDef*, *UsesDef*, and *EventPortDef* are defined as subclasses of the abstract metaclass *ComponentFeature* (see Figure 7.7):

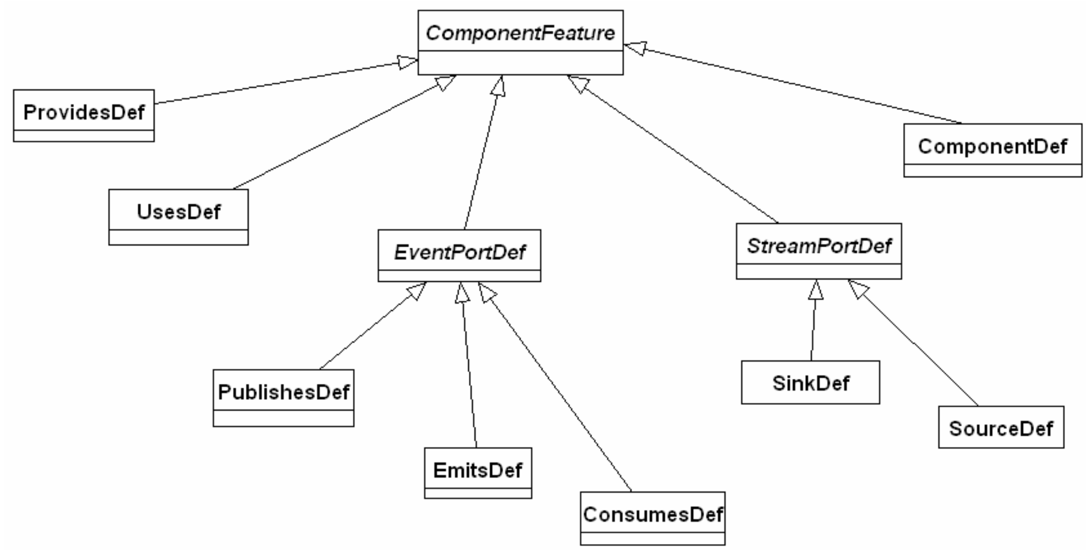

Figure 7.7 - ComponentFeature abstract metaclass

7.3.6 ComponentIDL Constraints

[9] A ProvidesDef can be defined only within a ComponentDef.

[9] context ProvidesDef inv:
self.definedIn.oclType = ComponentDef

[10] A UsesDef can be defined only within a ComponentDef.

[10] context UsesDef inv:
self.definedIn.oclType = ComponentDef

[11] An EventPortDef can be defined only within a ComponentDef.

[11] context EventPortDef inv:
self.definedIn.oclType = ComponentDef

[12] A FactoryDef can be defined only within a HomeDef.

[12] context FactoryDef inv:
self.definedIn.oclType = HomeDef

[13] A FinderDef can be defined only within a HomeDef.

[13] contextFinderDef inv:
self.definedIn.oclType = HomeDef

[14] A PrimaryKeyDef can be defined only within a HomeDef.

[14] context PrimaryKeyDef inv:
self.definedIn.oclType = HomeDef

[15] All of the ProvidesDef metaobjects that populate the Association component_facet also populate the ComponentDef's inherited Contains Association.

[15] context ProvidesDef inv:
component.contents->includesAll (facet)

[16] All of the UsesDef metaobjects that populate the Association component_receptacle also populate the ComponentDef's inherited Contains Association.

[16] context UsesDef inv:
component.contents->includesAll (receptacle)

[17] All of the EmitsDef metaobjects that populate the Association component_emits also populate the ComponentDef's inherited Contains Association.

[17] context EmitsDef inv:
component.contents->includesAll (emitss)

[18] All of the PublishesDef metaobjects that populate the Association component_publishes also populate the ComponentDef's inherited Contains Association.

[18] contextPublishesDef inv:
component.contents->includesAll (publishess)

[19] All of the ConsumesDef metaobjects that populate the Association component_consumes also populate the ComponentDef's inherited Contains Association.

[19] context ConsumesDe inv:
component.contents->includesAll (consumess)

[20] All of the SinkDef metaobjects that populate the Association component_sinks also populate the ComponentDef's inherited Contains Association.

[20] context SinkDef inv:
component.contents->includesAll (sinkss)

[21] All of the SourceDef metaobjects that populate the Association component_sources also populate the ComponentDef's inherited Contains Association.

[21] context SourceDef inv:
component.contents->includesAll (sourcess)

[22] All of the FactoryDef metaobjects that populate the Association home_factory also populate the HomeDef's inherited Contains Association.

[22] contextFactoryDef inv:
home.contents->includesAll (factories)

[23] All of the FinderDef metaobjects that populate the Association home_finder also populate the HomeDef's inherited Contains Association.

[23] context FinderDef inv:
home.contents->includesAll (finders)

[24] The ValueDef specified as the event type must descend directly or indirectly from Components::EventBase.

```
[24] contextValueDef  inv:
        type.descendsFrom ("Components::EventBase")

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
}
```

[25] The return type of FactoryDef must be the same as the type of the component that the FactoryDef's home manages.

[25] context FactoryDef inv:
self.type = home.manages.type

[26] The return type of FinderDef must be the same as the type of the component that the FinderDef's home manages.

[26] context FinderDef inv:
self.type = home.manages.type

[27] A ComponentDef C may be derived from at most one base.

[27] contextComponentDef inv:
self.base->size <= 1

[28] Furthermore, that one base must be a ComponentDef.

[28] context ComponentDef inv:
self.base->notEmpty implies (base->forAll (oclType = ComponentDef)

[29] A ComponentDef may not define operations.

[29] context ComponentDef inv:
self.contents->forAll (oclType <> OperationDef)

[30] A supported InterfaceDef of ComponentDef must not be one of the derived forms of InterfaceDef (i.e., ComponentDef or a HomeDef).

[30] context ComponentDef inv:
self.supports->forAll (oclIsTypeOf (InterfaceDef))

[31] A HomeDef may be derived from at most one base.

[31] context HomeDef inv:
base.size() <= 1

[32] Furthermore, that one base must be a HomeDef.

[32] context HomeDef inv:
base-> notEmpty implies (base->forAll (oclType = HomeDef)

[33] The valuetype of a primary key must not have private state members.

[34] The valuetype of a primary key must not have members that are interfaces.

[35] The valuetype of a primary key must have at least one state member.

[36] Contraints [33], [34], and [35] apply recursively to valuetype members that are valuetypes.

[33, 34, 35, 36] 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))

[37] Given a home definition H that manages a component type T, and given a home definition H' that manages a component type T', such that H' is derived from H, then T' must be identical to T or derived (directly or indirectly) from T.

[38] If H or one of its ancestors defines a primary key K and H' defines a primary key K', then K' must be identical to or derived (directly or indirectly) from K.

[37,38]

NOTE: Previously-defined additional OCL operation “descendsFrom” and new additional OCL operation “primaryKey” are used:

context HomeDef inv:
self.base->forAll (baseHome | self.manages.descendsFrom (baseHome.manages) and
primaryKey (self)->notEmpty implies
primaryKey (self).type.descendsFrom(primaryKey(baseHome).type))

primaryKey (home : HomeDef) : PrimaryKeyDef
{ if home.key->isEmpty then
if home.base->isEmpty then
result = home.key
else
primaryKey (home.base)
endif

```
else
        result = home.key
endi f }
```

[39] Basic ComponentDef objects shall not have ports and do not inherit from other components.

```
[39] context ComponentDef inv:
        self.isBasic implies
                facets->isEmpty and receptacles->isEmpty and
                emitss->isEmpty and publishess->isEmpty and consumess->isEmpty and
                sinkss->isEmpty and sourcess->isEmpty and
                base->isEmpty
```

[40] HomeDef objects of basic ComponentDef have only factories and finders, do not inherit from other homes, and manage only basic components.

```
[40] context HomeDef inv:
manages->isBasic implies (key->isEmpty and base->isEmpty and manages.isBasic)
```

[41] If StreamTypeDef object is a constructed stream, then its multiplicity must be more than one, in any other case the multiplicity is null.

```
[41] context StreamTypeDef  inv:
if self.kind = CONSTRUCTED_STREAM
        then
                groupedTypes.size() > 0
else
        groupedTypes.size() = 0
endif
```

[42] None of the StreamTypeDef of kind RAW_STREAM can be grouped.[42] context StreamTypeDef inv:

```
if groupedTypes.size() > 0
        then
                self.allInstances () -> forAll ( s | s.kind <> RAW_STREAM)
else
        groupedTypes.size() = 0
endif
```

7.4 CIF Metamodel

A CORBA Component encapsulates its internal representation and implementation. The Component Implementation Framework (CIF) metamodel defines the programming model for constructing component implementations described in Component Implementation Definition Language (CIDL). CIDL is a declarative language for describing the structure and state of component implementations (for more information please refer to the CORBA Component Model document formal/06-04-01). Component-enabled ORB products generate implementation skeletons from CIDL definitions. Component builders extend these skeletons to create complete implementations.

CIF metamodel package obviously depends on the ComponentIDL package (see Figure 7.1) since its main purpose is to enable the modeling of implementations for components specified using the ComponentIDL definitions. The extended metaclasses ComponentDef, HomeDef, and ComponentFeature from the ComponentIDL package are indicated with gray color in Figure 7.8.

The CIF metamodel represented in Figure 7.8 updates the metamodel defined in the CORBA Component Specification (formal/06-04-01). The updated CIF metamodel contains a new metaclass CompositionDef. This metaclass provides means for modeling component implementation as a composition of artifacts and will be explained in the next section.

To avoid unneeded complexity and misunderstanding metaclasses *ArtifactDef* and *Policy* have been deleted from the original CIF metamodel. To conform to composition definition in formal/06-04-01 (section 8.2.5) metaclasses *HomeImplDef* and *ComponentImplDef* have been renamed as *HomeExecutorDef* and *ComponentExecutorDef.*

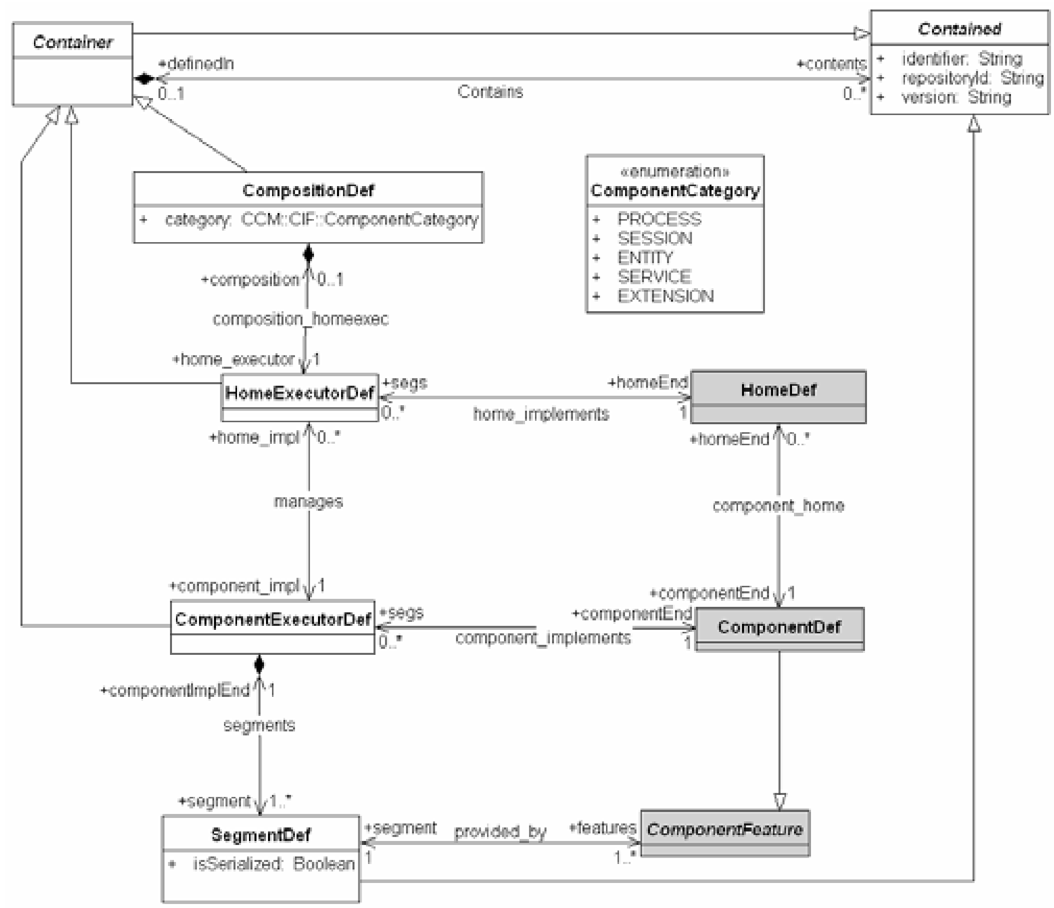

Figure 7.8 - Component Implementation Framework Metamodel

The term executor is used to indicate the programming artifact that supplies the behavior of a component or a component home. In general, the terms executor or component executor refer to the artifact that implements the component type, and the term home executor refers to the artifact that implements the component home.

CIF metamodel comprises a set of artifacts that must exhibit specific relationships and behaviors in order to provide a proper implementation. An overview on these is to be seen in Figure 7.8 and their meaning is explained in the following.

7.4.1 Composition

The description of a component implementation is a description of aggregate entity, of which the component itself may be a relatively small part. To denote the set of artifacts that constitute the unit of component implementation, the metaclass *CompositionDef* is defined. *CompositionDef* inherits from the *ContainerDef* metaclass and specifies the following metaclasses: *HomeExecutorDef*, *ComponentExecutorDef*, and *HomeDef* (from ComponentIDL metamodel). The name of the *CompositionDef* identifies the name of a scope within the contents of the composition (HomeExecutorDef, *ComponentExecutorDef*, and *HomeDef*) are contained. The attribute "*category*" of the *CompositionDef* identifies the life cycle category of the component implementation *ComponentExecutorDef* supported by the composition. The attribute has a type *ComponentCategory* defined as an enumeration type contained in five possible component categories: service, session, process, entity, and extension. The component categories service, session, process, and entity are defined and specified in formal/06-04-01, the extension category is added to the CIF metamodel for indication of vendor specific extensions done for a component implementation. For example, implementing the QoS extension can be done in a proprietary way by modifying the container. The QoS for CCM specification defines concepts for developing and integrating such extension for CCM in a standard way. The extended components differ from plain application components and should be deployed into containers of a particular type (container category). This type is the extension container type as defined in Section 5.9 of the QoS for CCM specification.

The most important properties of the component categories are briefly described below:

- Service: no state, no identity, behavior.
- Session: transient state, identity (which is not persistent), behavior.
- Process: persistent state, persistent identity, behavior, which may be transactional.
- Entity: persistent state, identity, which is architecturally visible to its clients through a primary key declaration, behavior, which may be transactional.
- Extension: vendor specific.

7.4.2 Component and Home Executors

The metaclass *ComponentExecutorDef* is used to model an implementation for a given component type. It specifies an association to *ComponentDef* to allow instances to point exactly to the component that the instance is going to implement. A *ComponentExecutorDef* always has exactly one ComponentDef associated while each ComponentDef might be implemented by different *ComponentExecutorDef* metaclass instances. *ComponentExecutorDef* is specified as being a Container, by doing so, instances are able to contain other definitions.

The *ComponentExecutorDef* definition optionally specifies executor segments (*SegmentDef* metaclass), which are physical partitions of the component executor, encapsulating independent state and capable of being independently activated. *Segments* are described in the next section. The only definitions that are allowed to be contained by a *ComponentExecutorDef* are instances of *SegmentDef*.

The metaclass *HomeExecutorDef* is used to model home executors (implementations). The name of the home executor is used as the name of the programming artifact (e.g., the class) generated by the CIF as the skeleton for the home executor. The contents of the *HomeExecutorDef* describe the relationships between the *HomeExecutorDef* and other elements of the composition, determining the characteristics of the generated home executor skeleton. Each instance of *HomeExecutorDef* in a model implements exactly one instance of *HomeDef*. This relation is modeled by the association implements between both metaclasses. *HomeExecutorDef* inherits from the abstract metaclass Container and manages exactly one *ComponentExecutorDef*, this relation is modeled by the association "*manages*."

7.4.3 Segments

A component implementation may be monolithic or segmented. A monolithic component implementation is a single artifact. A segmented component implementation is a set of physically distinct artifacts. Each segment may have a separate abstract state declaration. Each segment must provide at least one facet defined on the component definition. The life cycle category of the composition must be entity or process if the component implementation specifies segmentation. The primary purpose for defining segmented component implementations is to allow requests on a subset of the component's facets to be serviced without requiring the entire component to be activated.

The metaclass *SegmentDef* is used to model a segmented implementation structure for a component implementation. This means that the behavior for each facet (*ComponentFeature* abstract metaclass) can be provided by a separate segment of the component implementation (most likely a separate programming language class in the code generated by the CIF tools) if necessary. Instances of *SegmentDef* are always contained in instances of *ComponentExecutorDef* and therefore are derived from Contained. *SegmentDef* has an association to *ComponentFeature* so that instances must point to facets of a component that the segment is going to provide. The attribute isSerialized is used to indicate that the access to segment is required to be serialized or not.

The new CCM specification, version 4.0 obsoletes the original idea of component segmentation defined in pre-existing versions and allows composition and decomposition on any level, and therefore the ability to add another level of decomposition on the lowest level. This specification is based on the latest CCM specification and considers any level of component decomposition; this concept is defined in the Deployment and Configuration metamodel described in the next section.

7.4.4 CIF Constraints

There are no further additions or constraints on the CIF metamodel.

7.5 Deployment and Configuration Metamodel

Component implementations may be packaged and deployed. A Component package maintains one or more deployable implementations of a component. It may be installed on a computer or grouped together with other components to form an assembly. A component assembly is a group of interconnected components represented by an assembly package. Component and assembly packages are provided as input to a deployment tool. Based on deployment descriptors and user input, a deployment tool installs and activates component and home instances; it configures component instance properties and connects them together via interface, event, or stream ports.

The original CCM 3.0 Specification (formal/02-06-65) standardized deployment and configuration process for CCM applications in the section "Packaging and Deployment": deployment process steps, architecture, and deployment descriptors. Deployment descriptors are XML descriptions of component and assembly packages contents and other deployment information used by a deployment tool. One way for reducing the complexity of the deployment and configuration process of CCM components in the distributed environment is to have an expressive and robust metamodel and specific notation for modeling of deployment and configuration information, which can enable the automation of the entire deployment process of CCM applications (e.g., by generation of deployment descriptions automatically).

However, the original CCM specification defines neither a conceptual base for describing deployment and configuration requirements of components, nor a high level notation for the presentation of resulting models. The Deployment and Configuration of Component-based Distributed Applications (D&C) specification (formal/06-04-02) defines metadata and interfaces to facilitate the deployment and configuration of component-based applications into heterogeneous distributed target systems in general, in platform-independent manner. The original CCM 3.0 Specification (formal/02-06-65) has since been superseded by the new CCM 4.0 specification (formal/06-04-01), which describes mappings and extensions of the platform-independent model for Deployment and Configuration defined in formal/06-04-02 to CCM, but these mappings and extensions are not based on those defined in CCM 4 standard MOF metamodel. The reason was that at time of mapping definition some important concepts like component instance or implementation needed as conceptual base for deployment and configuration data definition were missing in the CCM metamodel. Thus, additional concepts for modeling of deployment and configuration data for CCM applications were defined and will be introduced in this section as a deployment metamodel. In that connection we tried to consider deployment information and concepts defined in both existing standards: D&C and CCM 4.0 and combine them together to provide a support for new deployment tools for CCM applications.

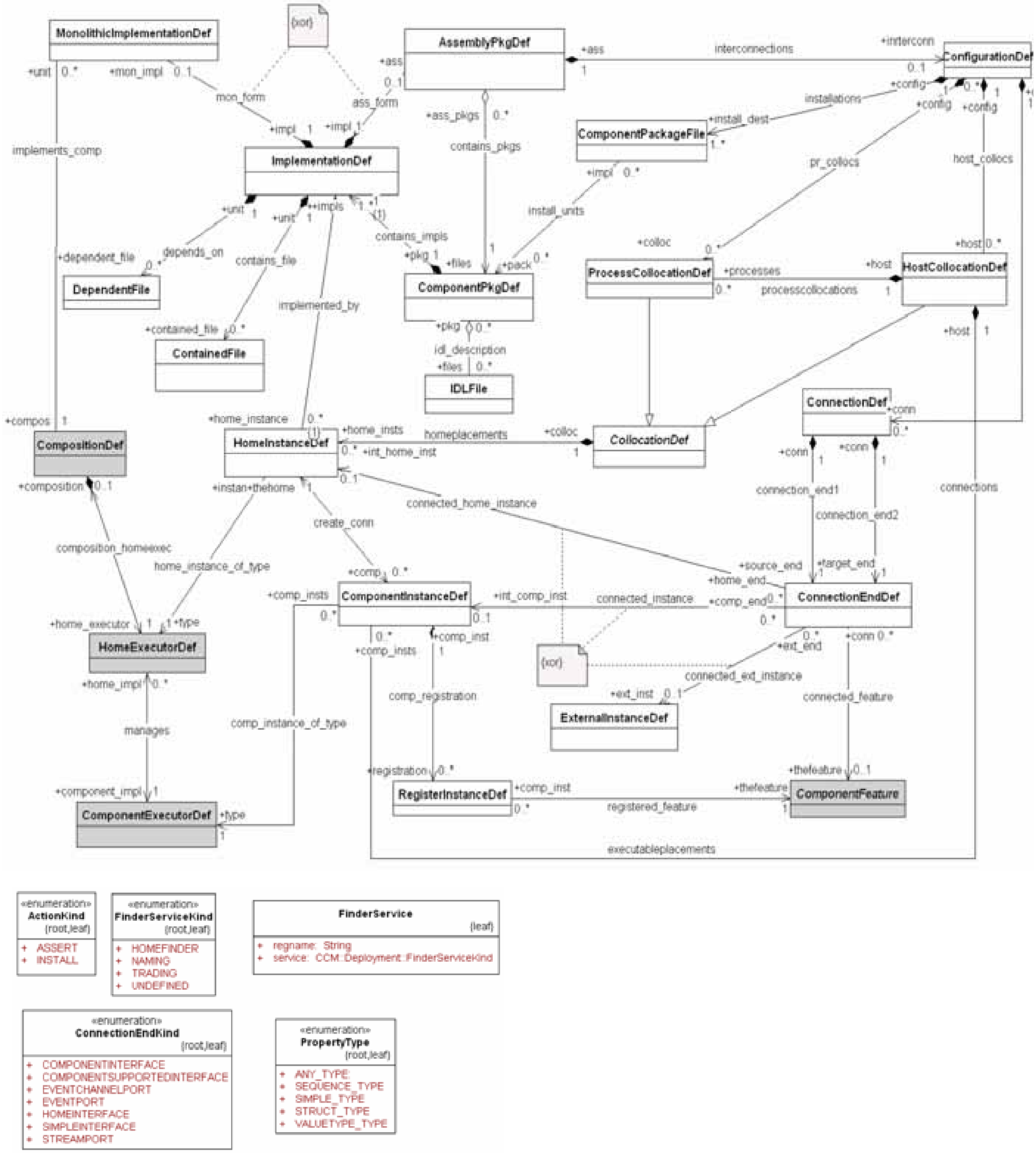

Figure 7.9 - Deployment metamodel: main diagram

The Deployment and Configuration metamodel defines a set of modeling elements represented as metaclasses and extends the existing CCM metamodel by these new metaclasses. In order to be able to model deployable CCM applications a concrete syntax must define the specific notation rules for the graphical representation of defined modeling language (metamodel). In our case we use UML 2: the UML Profile for Deployment and Configuration of CCM applications that will be introduced in the next section supports the representations of CCM deployment and configuration concepts in UML 2 models.

Deployment metamodel package depends on the CIF package (see Figure 7.1) since its main purpose is to enable the modeling of component packages and assemblies, which contain deployable implementation artifacts and instances of component and home implementations specified in CIF. Figure 7.9 shows all of the metaclasses and relationships defined in the Deployment metamodel package. The extended CIF metaclasses *ComponentFeature*, *CompositionDef*, *ComponentExecutorDef*, and *HomeExecutorDef* are identified with gray color.

7.5.1 Implementations

A component package of a CCM application represented by the metaclass *ComponentPkgDef* may contain a set of alternative implementations for one component (see association "*realized*" between *ComponentPkgDef* and *ComponentDef* metaclasses), for example, implementations for different operating systems, compilers, or ORBs, or different programming language implementations like JAVA or C++. These implementations, which contain descriptive information about a particular implementation of the software, are physical units of deployment process and represented by the metaclass *ImplementationDef*.

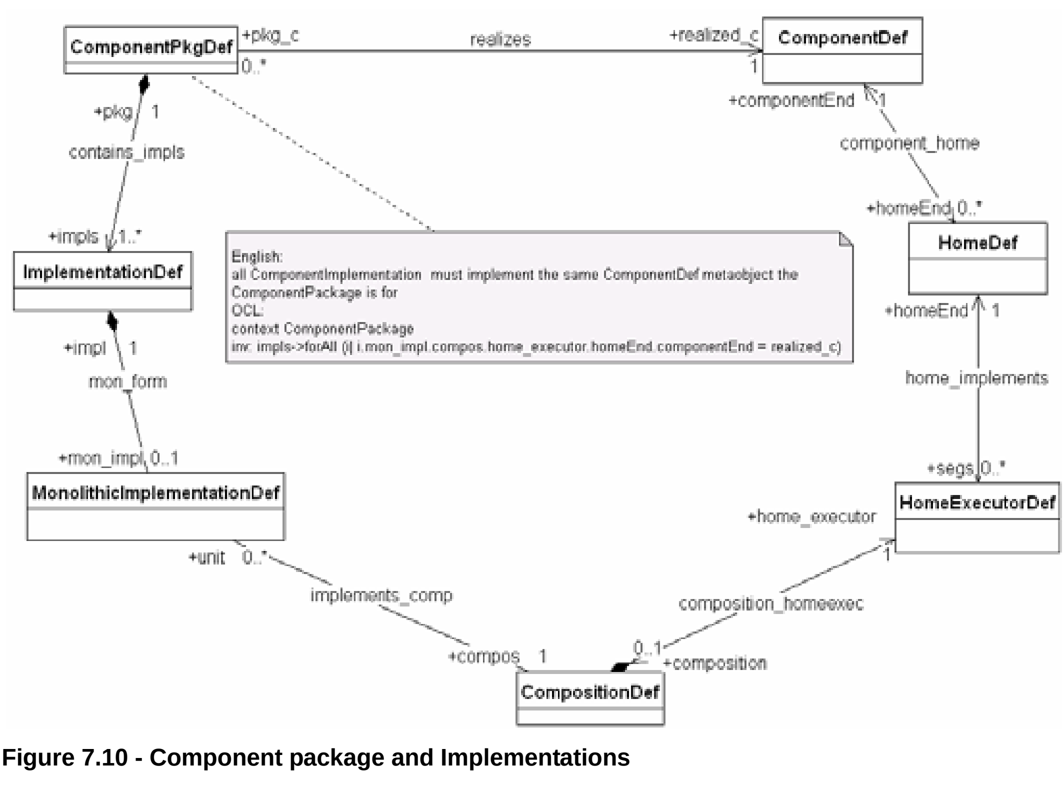

Figure 7.10 - Component package and Implementations

The Deployment & Configuration specification defines two types of component implementations: monolithic or assembly based implementations. A monolithic implementation is contained in an artifact (e.g., an executable file or library as a result), an assembly based implementation is a set of interconnected sub-component implementations. The monolithic implementation is represented by the metaclass MonolithicImplementationDef, assembly based implementation is represented by the metaclass *AssemblyPkgDef*.

The metaclass *CompositionDef* (see ComponentIDL metamodel) describes internal implementation structure of a *MonolithicImplementationDef*: for each monolithic component implementation one composition description must be defined.

The *ImplementationDef* metaclass is described by further *ContainedFile* and *DependentFile*, metaclasses, which are introduced below.

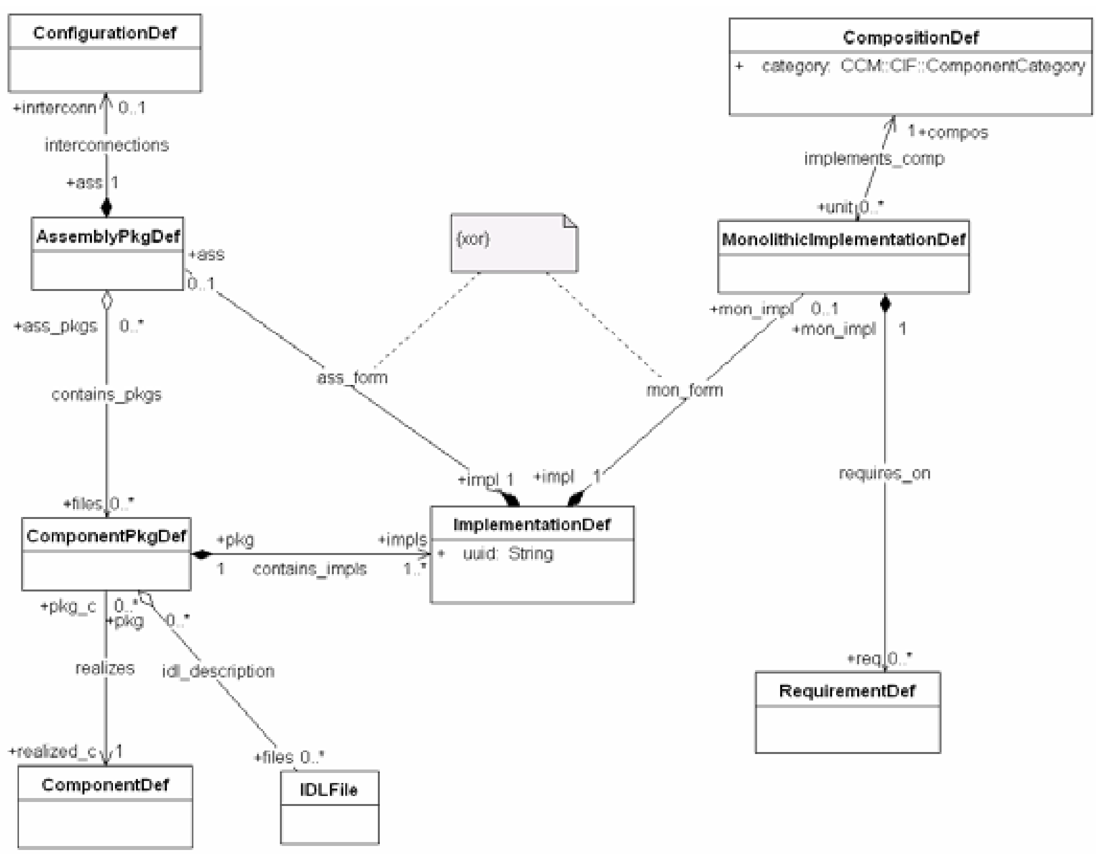

Figure 7.11 - ImplementationDef description

The *ImplementationDef* has the attribute “uuid” that uniquely identifies each instance of the *ImplementationDef* metaclass in a model. The *ImplementationDef* may have properties (e.g., configuration properties) or non-functional properties (e.g., QoS-properties), this feature the metaclass inherits from the metaclass *PropOwnerDef* (see Figure 7.13).

The *ComponentPkgDef* metaclass can point to the IDL file (metaclass *IDLFile*) containing an IDL definition of the component's (or home's) interface definition. A component package can be described by properties like author, title, or license information. These properties are defined using the abstract metaclass *PropOwnerDef* (see Figure 7.13).

The *RequirementDef* metaclass is used to specify features requested by component implementations (*ImplementationDef*) like compiler type, programming language, in which the *ImplementationDef* is realized; or type of operational system (os) that the ImplementationDef will work with. These features are described as properties, which the *RequirementDef* metaclass inherits from the abstract metaclass *PropOwnerDef* (see Figure 7.13).

The *ContainedFile* metaclass points to a file that implements the component, for example, a DLL or a .class file. The "*codetype*" attribute (Figure 7.14) specifies the type of code, the "*entrypoint*" attribute is used to specify an entry point to the code and the attribute "*entrypointusage*" is used to describe how to use (i.e., invoke) the code.

The *ContainedFile*, *IDLFile*, and *DependentFile* metaclasses are derived from the abstract metaclass File that defines general information about files like file name, file location, etc. (see Figure 7.14).

The *DependentFile* metaclass is used to specify environmental or other file dependencies of the ImplementationDef. When the attribute "*action*" is set to "*assert*" (see the enumeration *ActionKind* Figure 7.9), the installation process must verify that the dependency exists in the environment. If the attribute is set to "*install*," the installation process must install the dependency file if it does not already exist.

The *IDLFile* metaclass points to an IDL file. The IDL file is optional: some tools that deploy and execute CCM applications might need the IDL description to interact with the ports of the application's component interface.

7.5.2 Assembly Package

An assembly package as a main top object for deployment process. Assembly package may contain component packages and one description of the initial configuration of a CCM application. An initial configuration is a set of interconnected component implementation instances often called as an assembly; it is a template for instantiating a set of component implementations that make up the application and connecting them to each other at run time. This template or description is used by deployment tools as a main input.

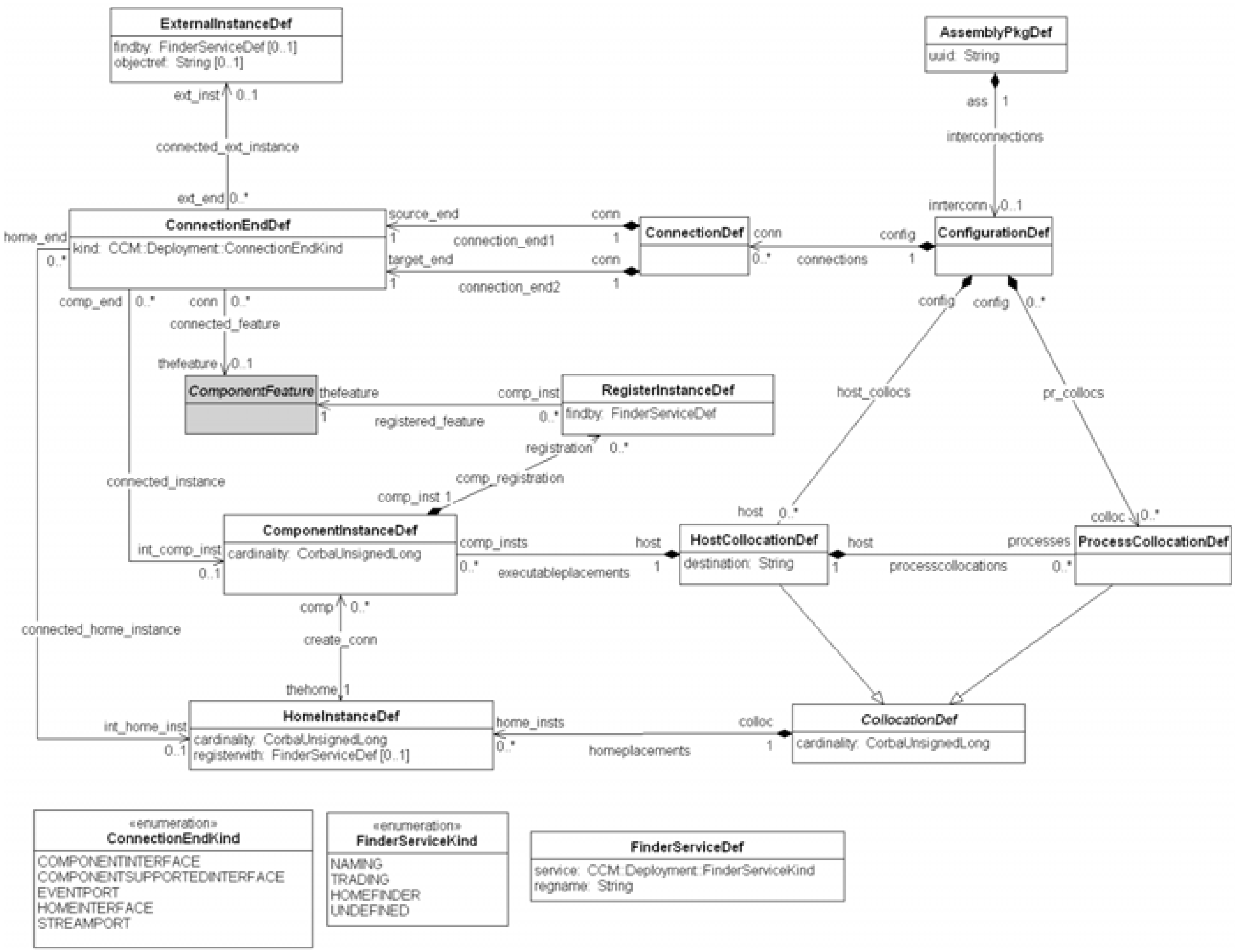

Figure 7.12 - Assembly package description

Figure 7.12 represents the assembly package description: the metaclass *AssemblyPkgDef* that uniquely is identified by the attribute "*uuid*" lists the component packages (*ComponentPackageDef*) that may be included in the assembly package and contain information regarding a component and home implementation. *AssemblyPkgDef* describes a possible initial configuration description of sub component implementation instances at the runtime (metaclass *ConfigurationDef*). The *ConfigurationDef* metaclass describes a template for instantiating of an assembly.

The *ConfigurationDef* metaclass specifies further metaclasses: *ProcessCollocationDef*, *HostCollocationDef*, *ComponentPackageFile*, and *Connection*.

Implementation instances of an application can be deployed either to one single host (host collocation) or different hosts; they may be executed in one single process (process collocation) or in different processes, which can be run on one or several hosts. The metaclasses *ProcessCollocationDef* and *HostCollocationDef* define these two different locality kinds. The abstract metaclass *CollocationDef* is a parent class for both metaclasses, its attribute "*cardinality*" specifies how many instances of the process or host collocation may be deployed. If the cardinality is greater than 1, and there are connections to components and homes within the collocation, then connections will be made to corresponding components or component homes within each instance of the collocation.

The *ProcessCollocationDef* metaclass specifies a group of home (*HomeInstanceDef* metaclass) and associated component (*ComponentInstanceDef* metaclass) instantiations that are to be deployed together to a single process.

A *HostCollocationDef* metaclass specifies a group of component instances that are to be deployed together to a single host.

The *ConnectionDef* metaclass describes how instances of deployed component and home implementations have to be initially connected to each other at the run time. The *ConnectionDef* is specified by two connection ends (*connection_end1* and *connection_end2* associations): one target and one source end. Both are described by the metaclass *ConnectionEndDef* and its attribute "*kind*," which indicates the kind of a connection end (*ConnectionEndKind*). There are several kinds of connection ends:

- COMPONENTINTERFACE is used to specify a connection between component uses and provides ports.
- COMPONENTSUPPORTEDINTERFACE is used to specify a connection between component with a supports interface and a uses port.
- HOMEINTERFACE is used to specify a connection between a home with an interface and a uses port.
- EVENPORT is used to connect a component consumes port to event producer.
- STREAMPORT is used to connect a component consumes port to stream producer.

The *ConnectionEndDef* is associated (association "*connected_feature*") with the abstract metaclass *ComponentFeature* defined in the ComponentIDL metamodel (see Figure 7.7) and generalized real component and component ports.

Each *Connection* connects two component instances via ports (*ComponentFeature* metaclass). The *ComponentInstanceDef* metaclass is used to describe the connected component instance created by its home instance (*HomeInstanceDef* metaclass). The attribute "*cardinality*" is used to specify how many instantiations of a component or home may be deployed. The attribute "*registerwith*" of the *HomeInstanceDef* instructs the installation process how to register the home and has the type of metaclass *FinderServiceDef*, which describes such register information.

The *RegisterInstanceDef* metaclass is used to specify that a component instance has to be registered after it is created. The attribute "*findby*" points to the registration kind (e.g., naming service or trader), its type *FinderServiceDef* is introduced below. The component type of the registered instance, provided interfaces, or published events are described by the metaclass *ComponentFeature*.

The *FinderServiceDef* metaclass is used to resolve a connection between two instances. Its attribute "*service*" tells how to locate a party, usually a component, interface, or home involved in the connection. In our case (see Figure 7.11), it could be located in a naming service (NAMING), in a trader (TRADING), by a home finder (HOMEFINDER), or by an undefined service (UNDEFINED).

7.5.3 Properties

The *PropertyDef* metaclass (Figure 7.13) specifies attribute settings of elements. Properties can be used at deployment time to configure home or component instances. Implementations may also have properties. The abstract metaclass *PropOwnerDef* specifies following property owners: *ComponentInstanceDef*, *HomeInstanceDef*, *AssemblyPkgDef*, *ComponentPkgDef*, *ImplementationDef*, *MonolithicImplementationDef*, and *RequirementDef*.

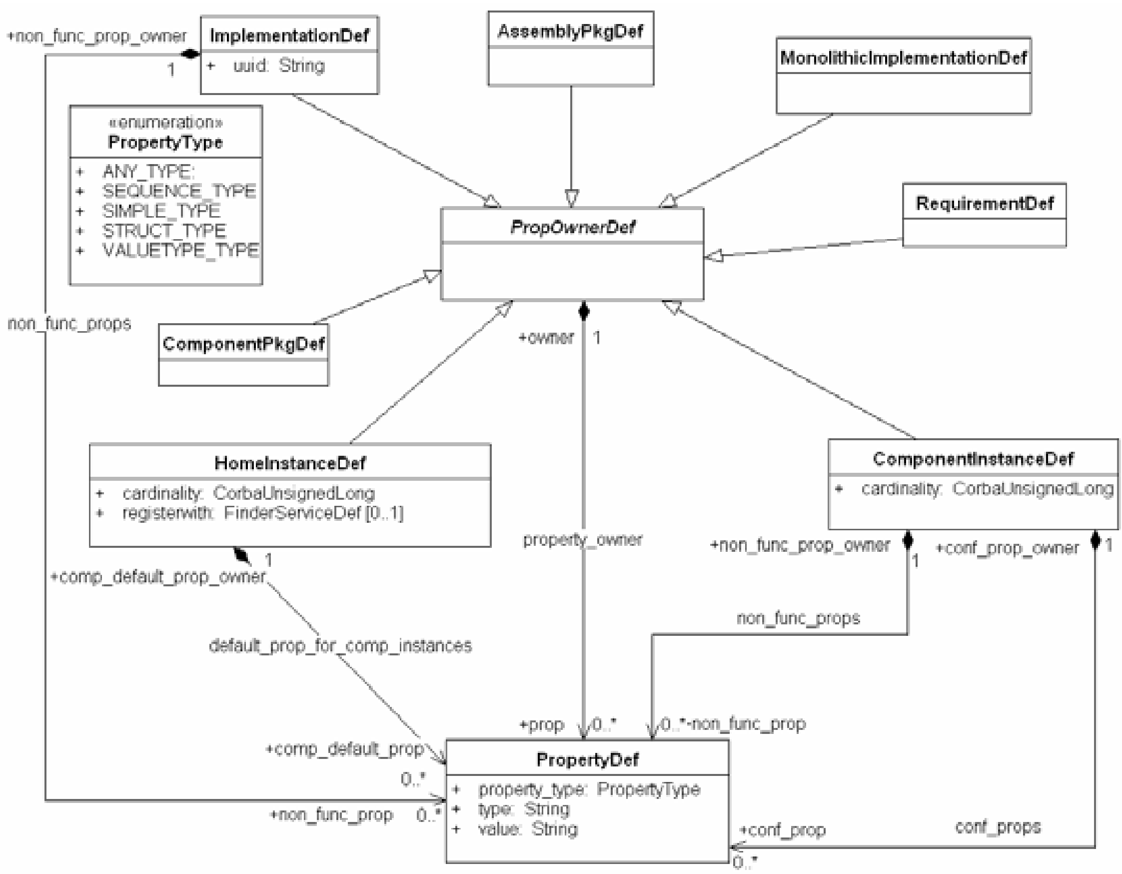

Figure 7.13 - Properties description

Each property has a property type defined as an *any* (ANY_TYPE), *simple* (SIMPLE_TYPE), *sequence* (SEQUENCE_TYPE), *struct* (STRUCT_TYPE), or *valuetype* (VALUE_TYPE) (see attribute "*property_type*" and the metaclass *PropertyType*). The simple property describes a single primitive BaseIDL type. The sequence property corresponds to a BaseIDL sequence, the struct corresponds to a BaseIDL struct, and the valuetype property corresponds to a BaseIDL valuetype.

7.5.4 Files

Component or Assembly packages may contain descriptors and a set of files. The descriptors describe the characteristics of packages and point to their various files. Figure 7.14 represents different file metaclasses: *ComponentPackageFile* points to component files, which are component packages, included in one assembly, *IDLFile* describes files that contain

IDL description, *DependentFile* specifies environmental or other file dependencies of an *ImplementationDef* metaclass, and *ContainedFile* (see also Section 7.5.1) specifies a file that contains the component implementation (e.g., DLL-File). All file metaclasses inherit from the abstract metaclass *File*, which has two attributes: "*filename*" and "*location*."

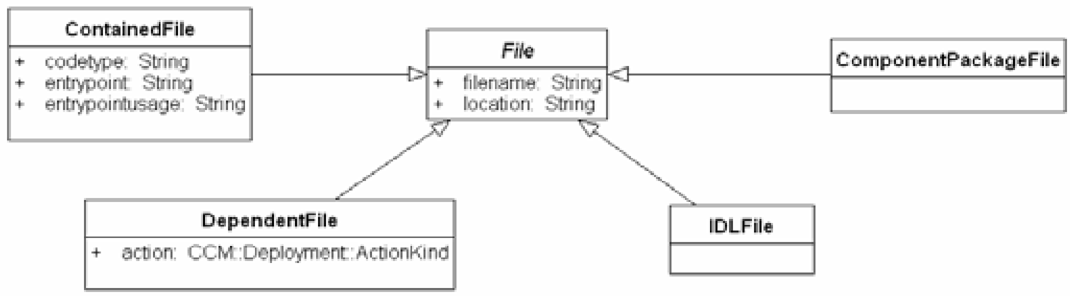

Figure 7.14 - File description

7.5.5 Containment

The following UML class diagram describes the derivation of the Deployment and Configuration metamodel elements from the BaseIDL Container and Contained elements:

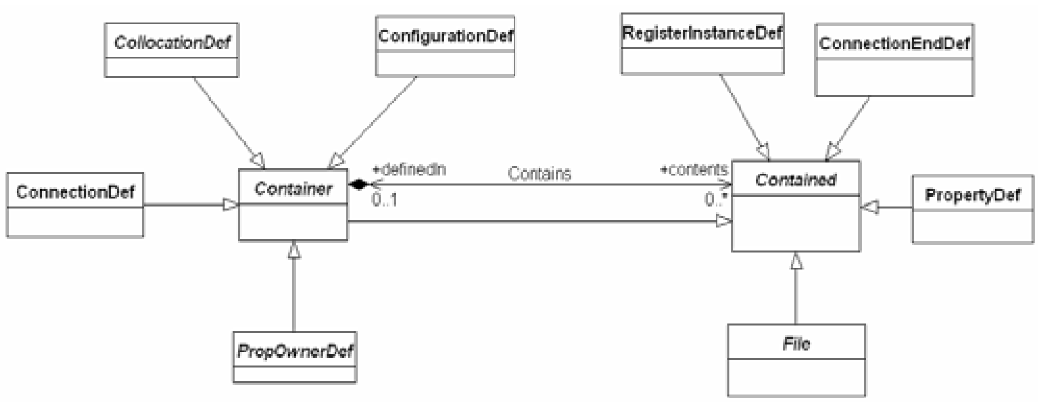

Figure 7.15 - Deployment and Configuration containment

7.5.6 Deployment Constraints

[49] A ComponentPkgDef may contain different Implementation objects that realize the same ComponentDef.

[49] context ComponentPkgDef inv:
impls->forAll (i| i.mon_impl.compos.home_executor.homeEnd.componentEnd = realized_c)

7.6 CCMQoS Metamodel

The modeling of non-functional properties such as QoS properties is defined in a platform independent way in the specification "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" (ptc/05-05-02). Chapter 8 of that specification defines a comprehensive metamodel for the description of QoS properties.

The modeling of QoS properties for CORBA Components requires the definition of a link between QoS metamodel and CCM metamodel. This link is defined in the metamodel package CCMQoS. The QoS metamodel consists of three packages:

1. QoSCharacteristics (defines the model elements for the description of QoS Characteristics),
2. QoS Constraints package (defines the modeling elements for the description of QoS contracts and constraints), and
3. the QoS Levels package (includes the modeling elements for the specification of QoS modes and transitions.

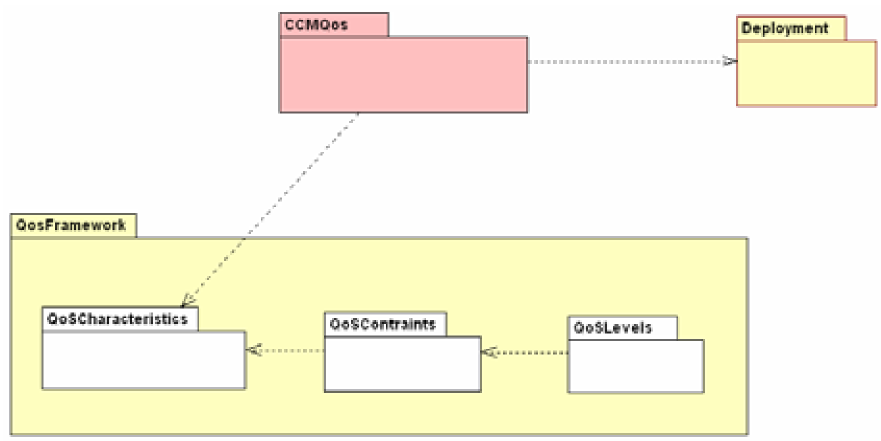

Figure 7.16 - CCMQoS package dependencies

The QoSCharacteristics metamodel defines a metaclass *QoSContext* that allows describing the context of quality expression (for more information please refer to ptc/05-05-02). We use this metaclass to describe QoS properties for CORBA Component and link it to the CCM metamodel (the Deployment metamodel is extended) by defining an additional metaclass *Binding*. The *Binding* metaclass has two attributes: “*name*” (the name of the *Binding*) and “*CCMQoS metamodel: Bindingmandatory*” (if “*true,*” then the QoS property is bound in any case).

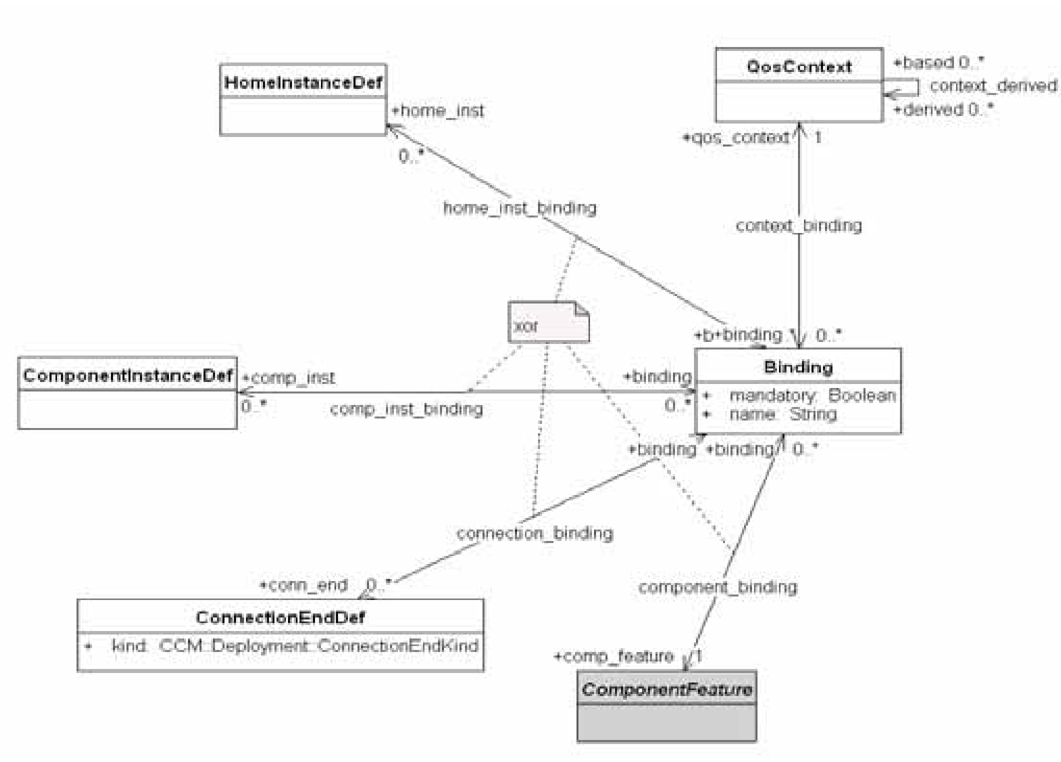

Figure 7.17 - CCMQoS metamodel: Binding

Due to the fact that definition of QoS properties for CORBA Components may have different scopes, different links for QoS properties need to be defined. So, the metaclass Binding (see Figure 7.17) correlates a *QoSContext* with a component feature definition (*ComponentFeature*), or a *ComponentInstanceDef*, or a *HomeInstanceDef*, or a *ConnectionEndDef*. Binding the *QoSContext* to the *ComponentFeature* makes QoS property applicable for the component type. This means that also all instances of this component type are related to the *QosContext*. The binding of the *QoSContext* to the *ComponentInstanceDef* makes QoS properties relevant to only a specific component instance but for the component instance in general. The binding the *QoSContext* to the *HomeInstanceDef* makes QoS properties relevant to a group of component instances that are managed by a specific home instance. The binding of the *QoSContext* to a *ConnectionEndDef* make QoS properties only relevant to a specific port of a component instance.

8 UML Profile for CORBA and CORBA Components

The UML Profile for CORBA and CORBA Components (CCM profile) defines limited extensions to the reference UML2 metamodel with the purpose of adapting the UML metamodel to the CORBA Components. The extension done by this profile does not change the UML2 metamodel, and keeps its semantics.

In UML2, profiles are packages that structure UML extensions. The principal extension mechanism in UML2 is the concept of stereotype. Stereotypes are specific metaclasses, having restrictions and the specific extension mechanism.

Additional semantics can be specified using Stereotype properties (“attributes” in UML2, "tagged values” in UML1.x) and constraints in the context of a profile.

A UML profile extends parts of the UML metamodel in a constrained way. All new modeling concepts must be supported by UML modeling elements. The new attributes must respect the semantic of UML modeling elements. All associations are binary associations. We are not able to redefine features, but we can add new features (meta attributes of stereotypes). UML metaclasses are extended by stereotypes, using a mechanism called extension.

For the Profile specification we use both graphical and tabular notations. To be able to interchange CCM Profile between tools, together with models to which they have been applied, the Profile is defined as an interchangeable UML model (by using the UML XMI interchange mechanisms):

- The metaclass extensions are expressed via UML class diagrams.
- A Profile is a kind of UML Package that extends the UML metamodel.
- A stereotype is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. Each stereotype is expressed via a stereotyped with <<stereotype>> Classifier box. All classes, which define stereotypes and extend the UML metamodel, are indicated with yellow color; all original metaclasses from the UML metamodel are indicated with white color on represented following class diagrams.
- When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values.
- Like a class, a stereotype may have properties that may be referred to tag definitions.
- Each stereotype is a client in a UML Extension with the UML metaclass that it extends. This Extension (UML Association) is stereotyped with <<extends>>.
- Generalization Relationships among stereotypes are expressed in the standard UML manner.

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 and in parenthesis “()” the name of the metaclass or association between two metaclasses from the CCM metamodel (Profile to Metamodel mapping), which instances are represented by this stereotype in UML models.
- Base Class: the UML metamodel element that serves as the base for the stereotype.
- Parent: the direct parent of the stereotype being defined (Note: 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).
- Constraints: a list of constraint numbers applied to the stereotype.

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 operations are used in the CCM Profile specification; they were defined in the UML1.3 Profile for CORBA and adopted for this specification in order to produce more compact and readable OCL:

For Element

[1] The operation allStereotypes results in a Set containing the Element's Stereotype and all Stereotypes inherited by that Stereotype (as opposed to all Stereotypes inherited by the Element).

context Element inv:
allStereotypes : Set(Stereotype);
allStereotypes = self.stereotype->union (self.stereotype.generalization.parent.allStereotypes)

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

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

[3] The operation isStereokinded determines whether the ModelElement has a Stereotype whose name is equal to the input name or if it has a Stereotype one of whose ancestors' name is equal to the input name.

context Element inv:
isStereokinded : (stereotypeName : String) : Boolean;
self.allStereotypes->exists (stereotype | stereotype.name = stereotypeName)

Some abstract Stereotypes are defined and, in keeping with UML notation, abstractness is denoted by italicizing the Stereotype's name; they cannot be instantiated. The abstract Stereotypes are useful for avoiding repetition in multiple Stereotypes that logically have common properties.

Profile Structure

The general structure of the CCM profile model is the same as the general structure of the CCM metamodel and is shown in Figure 8.1.

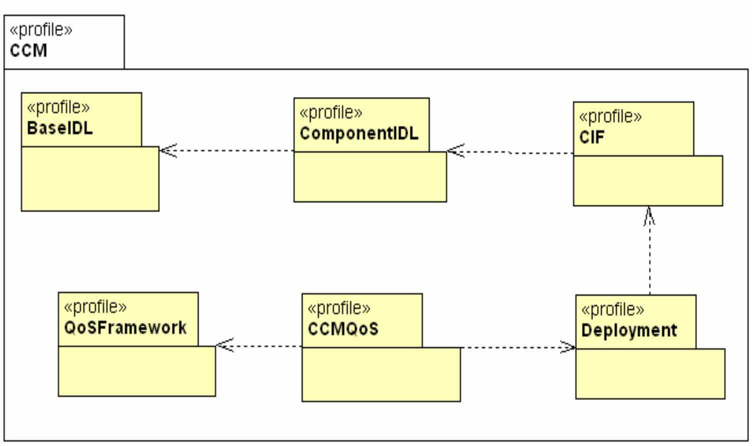

Figure 8.1 - CCM Profile package structure

8.1 BaseIDL Profile

This chapter is the normative definition of the CORBA plain (BaseIDL) Profile of UML. It consists of a UML model, showing extensions to UML (stereotypes) using the notation described in the previous chapter. This is followed by a tabular description of the Profile and defined constraints.

8.1.1 CORBA Module, Interface, Value, Constant Stereotypes

An IDL module is represented by a UML package (from Kernel) stereotyped as <<CORBAModule>>. IDL module containment (nesting) is modeled by Namespace containment of one <<CORBAModule>>-stereotyped UML package within another.

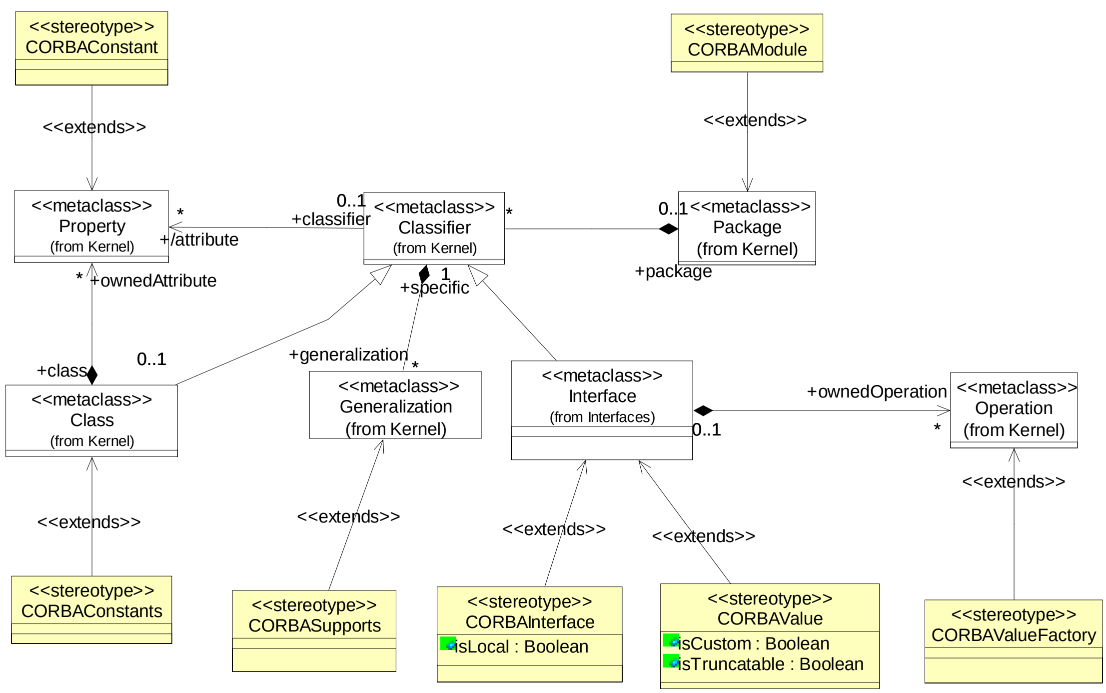

Figure 8.2 - BaseIDL Profile: Extended UML classes (I)

CORBA interfaces are represented using a UML Interface (from Interfaces) stereotyped as <<CORBAInterface>>. Local interfaces are represented using the tagged value "*isLocal*" = TRUE.

CORBA value types are represented by a UML Interface stereotyped as <<CORBAValue>>. CORBA custom value types are represented by the tag "isCustom" and truncatable value types are represented by the tag "*isTruncatable*."

CORBA interfaces and value types may have attributes and operations.

Attributes are represented as UML Properties (Attributes) - as usual in UML, each IDL operation is represented as a UML Operation. A value type factory operation, which is some kind of constructor, is represented using a UML Operation that is stereotyped <<CORBAValueFactory>>.

Values may be derived from other values and can support an interface. The support by a value type of an IDL interface type is represented by a Generalization relationship, which is stereotyped <<CORBASupports>>.

An IDL constant is modeled as a stereotyped with <<CORBAConstant>> UML Property, with the constant value expression represented by the Property's attribute "*default*" (default: String [0..1]).

For constants defined within a CORBA module scope a new stereotype <<CORBAConstants>> for UML Class is introduced. The name of the Class must be "***Constants***."

The UML notation of a CORBA module for the following IDL example is shown in Figure 8.3.

```
module Parent
{
      module Child1 {};
      module Child2
      {
            module Grandchild {};
      };
};
```

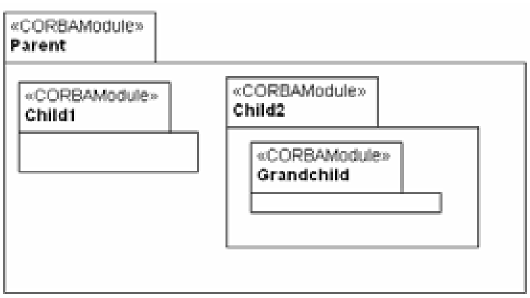

Figure 8.3 - CORBA Module notation

The UML notation of an interface for the following IDL example is shown in Figure 8.4:

```
interface TestInterface
{
    struct TestStruct
    {
          string Member1;
    };
    attribute string MyStringAttr;
    attribute TestStruct MyStructAttr;
    void MyOp1( in string str, inout TestStruct t);
    boolean MyOp2( inout TestStruct t);
};
```

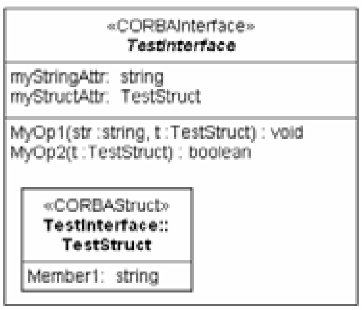

Figure 8.4 - Example Interface containing a Struct

The UML notation for a CORBA value type for the following IDL example is shown in Figure 8.5:

```
interface PrettyPrint
{
       string print();
};
valuetype Time
{
       public short hour;
       public short minute;
};
valuetype Date
{
      public short day;
      public short month;
      public short year;
};
valuetype DateAndTime : Time supports PrettyPrint
{
       private Date the_date;
       factory init( in short hr, in short min);
       Date get_date();
};
```

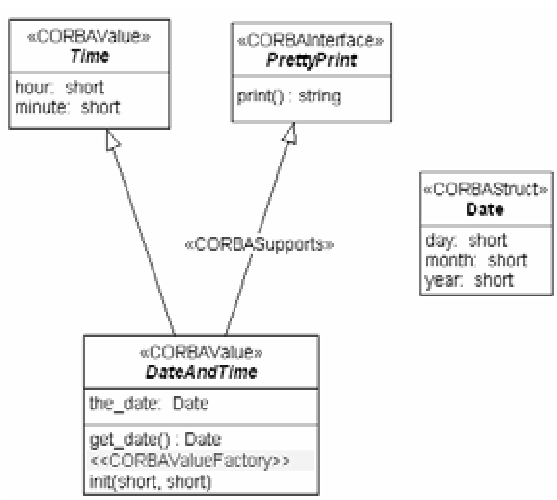

Figure 8.5 - Valuetype example

The UML notation for CORBA constants for the following IDL example is shown in Figure 8.6.

```
module Y
{
      constant short S = 3;
      interface X
      {
            constant long L = S + 20;
      };
};
```

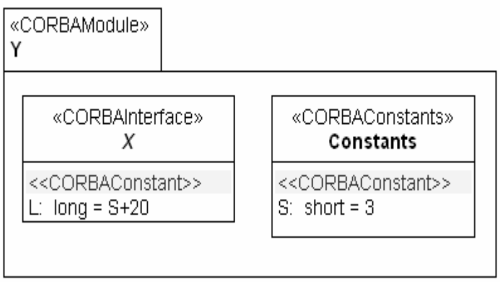

Figure 8.6 - Constant example

8.1.2 Other stereotypes: CORBA Types

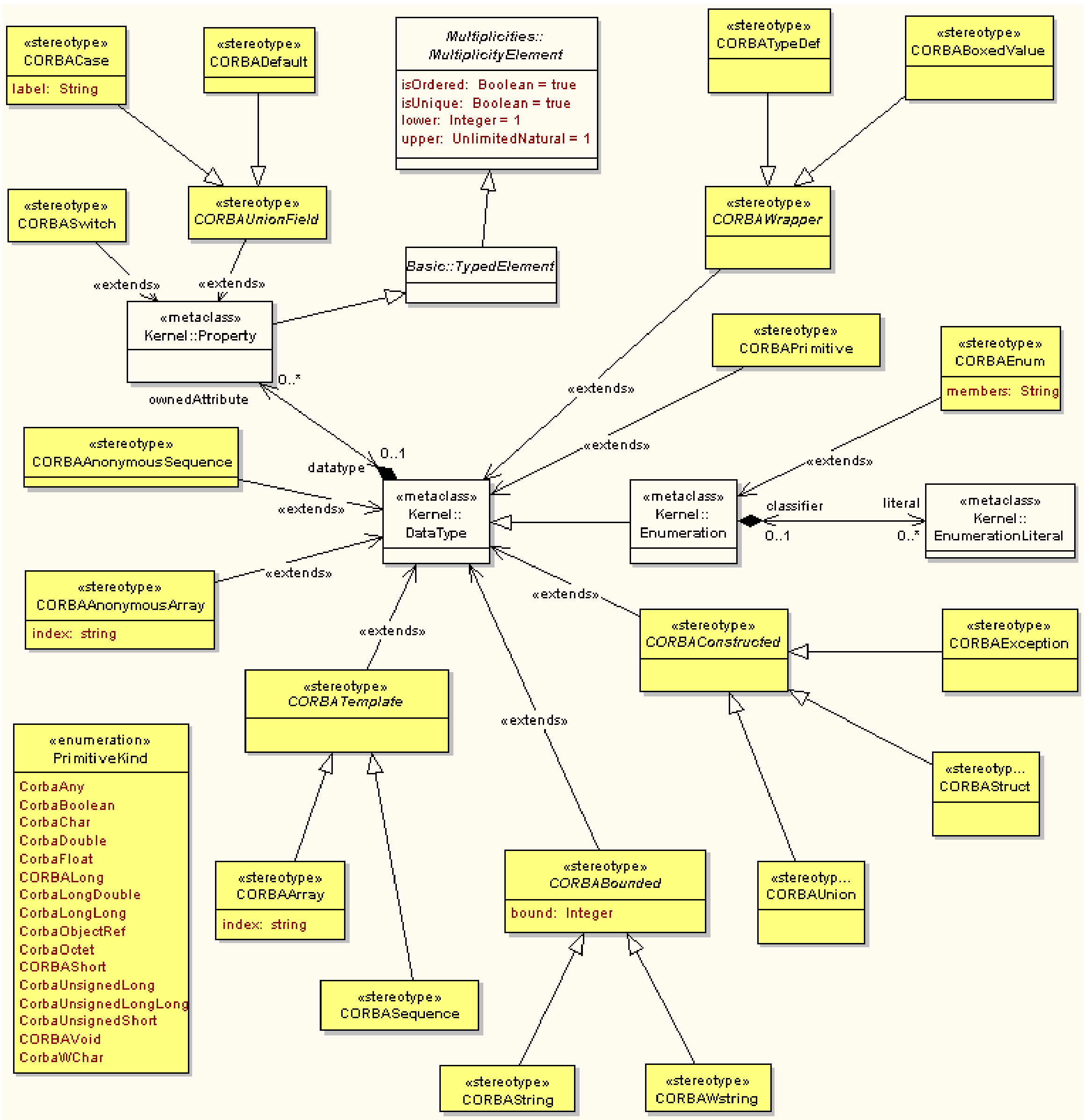

Figure 8.7 - BaseIDL Profile: Extended UML classes (II)

A UML DataType (from Kernel) is a type whose instances are identified only by their value. A DataType is a special kind of classifier, similar to a class and may contain attributes to support the modeling of structured data types.

CORBAPrimitive

The CORBA basic and other types are represented by the UML DataType with the <<CORBAPrimitive>> stereotype in the "CORBA" package. This package also contains the base types for CORBA interfaces and value types.

The following <<CORBAPrimitive>>-stereotyped UML DataTypes are introduced in the package “CORBA.” their semantics is defined in the “OMG IDL Syntax and Semantics” chapter of the CORBA Specification.

- short
- long
- longLong
- double
- longDouble
- unsignedShort
- unsignedLong
- unsignedLongLong
- any
- boolean
- octet
- void
- char
- wchar
- float
- string
- wstring

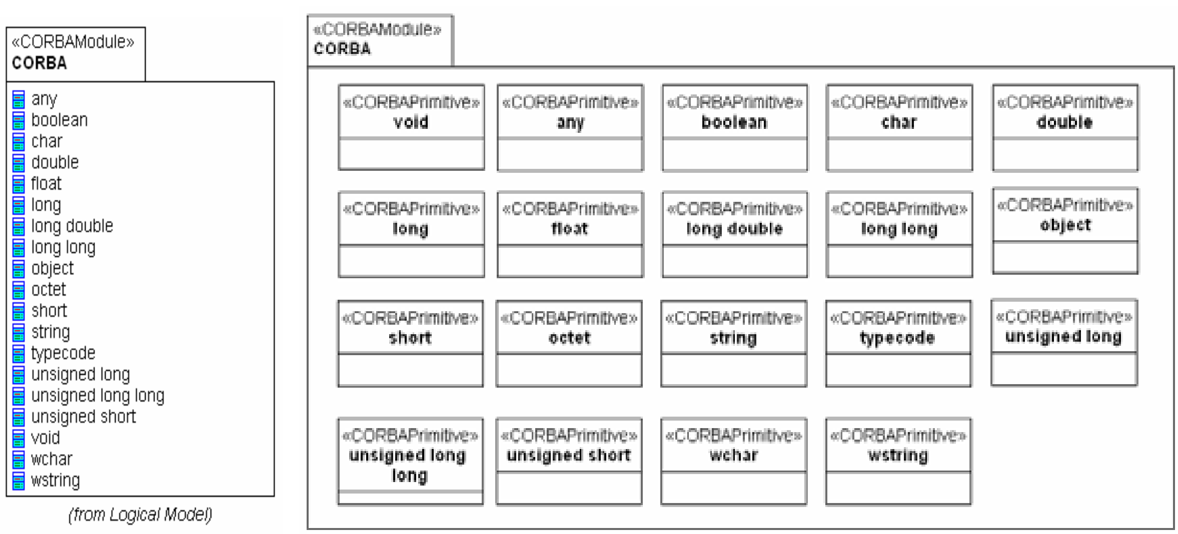

Figure 8.8 - CORBA package

CORBAConstructed

The extended UML metamodel contains an abstract stereotype <<CORBAConstructed>>, which is a generalization of <<CORBAStruct>>, <<CORBAUnion>>, and <<CORBAException>> stereotypes. Each of them shares the characteristics of having ordered named elements of some CORBA type. Each member of a constructed type that is either of a CORBA basic type or user defined type is represented as a UML Property (Attribute) by the Data Type. The order of members is represented by the attribute "*isOrdered*" of the UML Property class, which inherits this attribute from the metaclass MultiplicityElement (see Figure 8.7). This attribute specifies whether the values in an instantiation of this element are ordered. Default value is true.

CORBAEnumeration

IDL enumerations consist of ordered lists of identifiers and are represented by UML Enumeration stereotyped as <<CORBAEnum>> whose values are enumerated in the model as enumeration literals. An example of <<CORBAEnum>> contents is represented in Figure 8.9.

The type and initial numeric values of the UML Attributes representing enumeration elements may be omitted in the notation, as the type is always short, and the initialValue can be deduced from the ordering of the Attributes.

CORBAUnion

IDL union definitions are represented by a UML DataType stereotyped as <<CORBAUnion>>.

Each member of the IDL union is represented by the abstract class CORBAUnionField that extends a UML Property. The abstract stereotype <<CORBAUnionField>> is specialized to the concrete stereotypes <<CORBADefault>> and <<CORBACase>>.

The discriminator type is represented as an additional Property of the <<CORBAUnion>>, which is stereotyped as a <<CORBASwitch>>. This Property has always the name "*discriminator.*" Case labels are referred to the defined type of the discriminator. Each member of an IDL union is represented as a UML Property (Attribute) and stereotyped either <<CORBADefault>> or <<CORBACase>>. <<CORBACase>>-member has a Tag "*label*" attached with its label name value being the case label for this member in the union declaration. For union declarations in which there is a default case, the <<CORBADefault>>-member is used.

```
enum Contents
{
       INTEGER_CL;
       FLOAT_CL;
       DOUBLE_CL;
       COMPLEX_CL;
       STRUCTURED_CL;
};
union Reading switch (Contents)
{
       case INTEGER_CL: long a_long;
       case FLOAT_CL:
       case DOUBLE_CL: double a_double;
       default: any an_any;
};
```

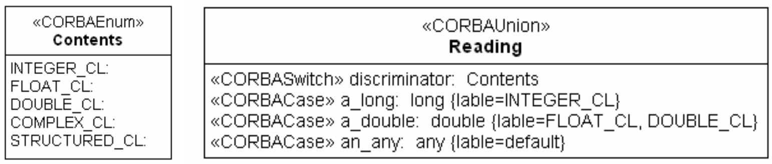

Figure 8.9 - Union example (a)

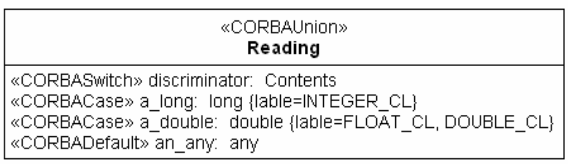

Figure 8.10 - Union example (b)

CORBAStruct

IDL struct definitions are represented by a UML DataType stereotyped as <<CORBAStruct>>. Each member of the IDL struct can be represented as a UML Property (Attribute) as shown in Figure 8.11 and Figure 8.12.

<<CORBAStruct>> inherits from the abstract stereotype <<CORBAConstructed>> described above and defines a new name scope containing other declarations (members). These members must have the same order as in derived IDL declarations (when UML models are derived from IDL) to be able to generate correct equivalent IDL from the UML model.

```
struct Fraction
{
      double numeric;
      string alphabetic;
};
struct Problem
{
      string expression;
      Fraction result;
      Boolean correctness;
};
```

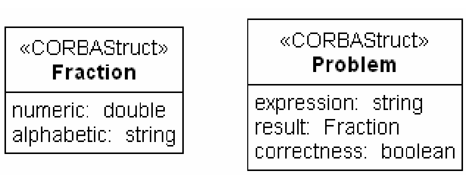

Figure 8.11 - Struct example

A CORBA struct can act as the namespace (see Figure 8.7 metaclass Namespace) for the following CORBA types: structs, unions, and enums. Only these three types can be defined within struct's scope. Nesting of elements limits the visibility of the element to within the scope of the namespace of the containing struct and is used for reasons of information hiding.

```
struct A
{
        struct B
        {
                short k;
                long j;
        } p;
        string q;
};
```

The following example (IDL above) demonstrates alternative UML representations (dependent on the UML tool) of the nested struct B:

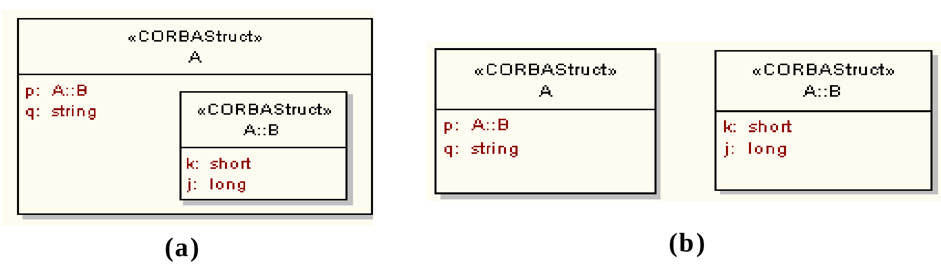

Figure 8.12 - Alternative Struct representations with nested elements

CORBAException

IDL exception definitions are represented by UML DataType stereotyped as <<CORBAException>>. Each member of the IDL Exception is represented as a UML Property. Exceptions, like structures, create a namespace, so the exception member names need to be unique only within their enclosing exception. Exceptions are types but cannot be used as data members of user-defined types.

The following IDL can be represented in UML as in Figure 8.13.

```
exception Failed {};

exception RangeError
{
       unsigned long supplied_val;
       unsigned long min_permitted_val;
       unsigned long max_permitted_val;
};
interface Unreliable
{
      void can_fail() raises (Failed);
      can_also_fail ( in long l) raises ( Failed, RangeError );
};
```

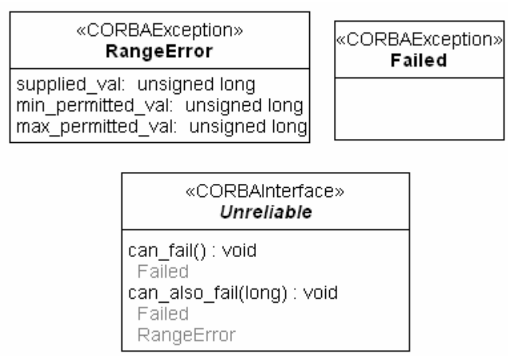

Figure 8.13 - Exception example

The extended UML metamodel contains an abstract stereotype <<CORBATemplate>>, which is a generalization of <<CORBAString>>, <<CORBAWstring>>, and <<CORBASequence>> stereotypes. All <<CORBATemplate>> elements have a tag "*bound*" that indicates the maximum size of the element.

CORBAString and CORBAWstring

IDL string is similar to a sequence of char and represented by a UML DataType stereotyped as <<CORBAString>>. IDL *wstring* is like a sequence of wchar and represented by a UML DataType stereotyped as <<CORBAWstring>>. The type *wstring* is similar to *string*, except that its element type is *wchar* instead of *char.* If a positive integer maximum size is specified, the *string* (or *wstring*) is termed a bounded *string* (or *wstring*); if no maximum size is specified, the string (or *wstring*) is termed an unbounded *string* (or *wstring*). The package "CORBA" (see Figure 8.8) contains unbounded *string* and *wstring* elements (no maximum size is specified) as stereotyped <<CORBAPrimitive>> UML DataTypes. Bounded IDL strings and wstrings are represented by a UML DataType stereotyped as <<CORBAString>> or <<CORBAWstring>>.

CORBASequence

A CORBA Sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time) and a length (which is determined at run time).

CORBA Sequences are IDL template types that take a CORBA type as their element parameter, and optionally an integer as an upper bound specification. Sequences are anonymous, and can either be named by a typedef, or by the member name of a constructed type. Sequences are represented in the Profile by two means:

- Named by a typedef declaration sequences are represented by the UML DataType with the stereotype <<CORBASequence>>. Sequence members are represented by an attribute of the DataType, which always has the name “members” (profile keyword), members type is represented by the type of the “members”-attribute and the max size is represented by the multiplicity of the “members”-attribute.
- Named by the member name of a constructed type sequences are represented by the UML DataType with the stereotype <<CORBAAnonymousSequence>>. Anonymous sequences get a name by concatenation name of the container (containing type), “::” and “m”<n>, where n is the member number of the anonymous sequence in the container.

Sequences that are declared as the type-declarator of a typedef are given the name of that typedef and the stereotype <<CORBASequence>>. The following IDL example is represented in Figure 8.14.

```
typedef sequence<short, 4> foo;
typedef sequence<long> foo_lg;
```

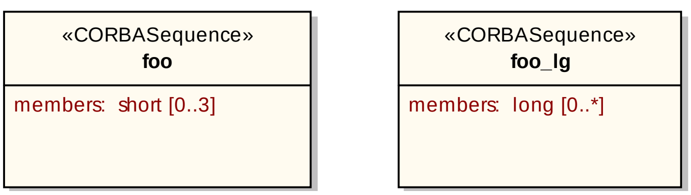

Figure 8.14 - CORBASequence example

Sequences that are anonymous (declared in some context where they don't have a type name, such as a struct member type) are given the stereotype <<CORBAAnonymousSequence>>. The following IDL example has the sequence declaration as a struct member. The UML notation for this example is represented in Figure 8.15.

```
struct bar
{
      long val;
      sequence <short, 6> my_shorts;
};
```

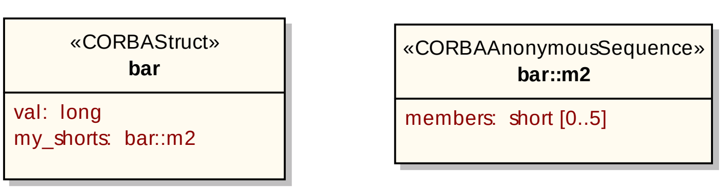

Figure 8.15 - CORBAAnonymousSequence example

The following IDL is featuring an anonymous sequence as the type of another sequence is represented below.

```
typedef sequence <sequence <string,4>> foo_1;
```

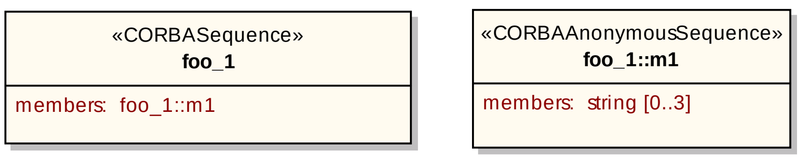

Figure 8.16 - Nested CORBAAnonymousSequence example

CORBAArray

OMG IDL defines multidimensional, fixed-size arrays. CORBA Arrays are IDL indexed types that take a CORBA type as their element type, and have at least one integer as the size of the array. Additional array dimensions are specified by additional integers. Arrays are anonymous, and can either be named by a typedef, or by the member name of a constructed type. Similar to sequences arrays are represented in the Profile by two means:

- Named by a typedef declaration arrays are represented by the UML DataType with the stereotype <<CORBAArray>>. Array members are represented by an attribute of the DataType, which always has the name “members” (profile keyword), “members” type is represented by the type of the attribute. The array size (dimensions) is represented by the tag “index” of the DataType. The value of the “tag „index” is a list of integers separated by comma where each integer represents the size of each multidimensional array dimension (e.g., “index”=n, m). One-dimensional arrays are represented as “index”=n, where n is an integer and must be greater than 0.
- Named by the member name of a constructed type arrays are represented by the DataType with the stereotype <<CORBAAnonymousArray>>. Anonymous arrays get a name by concatenation name of the container (containing type), “::” and “m”<n>, where n is the member number of the anonymous array in the container. This stereotype has also the tag “index” representing the index of the array described above.

Since CORBA IDL does not support open arrays like “typedef short s [];” because IDL does not support pointers, integer m, n, and k must be greater than 0.

CORBA IDL array determines the number of elements of an array, but IDL does not specify how elements of the multidimensional arrays are to be ordered for data transfer between agents. Therefore, for common and correct understanding of CORBA UML models the same convention as GIOP (defined in the CORBA Specification) is used: the array members represented by the “members” attribute are always in *row major* order. In row major ordering the leftmost index (or index of the first dimension) varies most slowly, and the rightmost index (or index of the last dimension) varies most quickly.

Arrays that are declared as the type-declarator of a typedef are given the name of that typedef and the stereotype <<CORBAArray>>. The following IDL example shows UML representations for arrays.

```
typedef short short_arr[4];
typedef my_struct my_struct_arr[5][10];
```

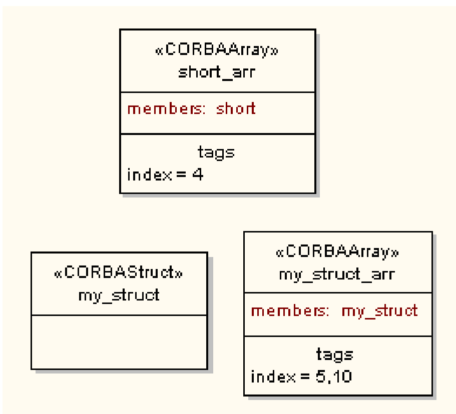

Figure 8.17 - Array example declared as typedef

An IDL array that is declared in any other context is represented by an Attribute stereotyped as <<CORBAAnonymousArray>>. The following IDL is represented in Figure 8.18.

```
struct boom
{
        string zoom[4];
        my_struct loom[2][2][2];
};
```

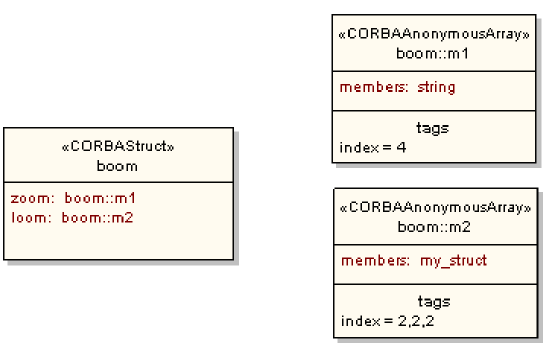

Figure 8.18 - Anonymous Array representation

There are two declarations in IDL that provide existing named types with another identifier:

- typedefs - give a name to an existing type (or to a new template type), and
- boxed value declarations - give a new name to an existing type, and allow the new type to be passed as a null parameter.

Such declarations are called “wrapper” declarations and represented by the abstract stereotype <<CORBAWrapper>>. There are two concrete specializations of <<CORBAWrapper>>: <<CORBATypedef>> and <<CORBABoxedValue>>.

CORBATypeDef

Typedefs in IDL serve two purposes. The first purpose is to rename types that already have names to provide an alias for an existing type. These typedefs are represented by UML DataTypes stereotyped as <<CORBATypeDef>>. For example, the IDL below provides an alias “Alias_Interface” for the interface “Initial_Interface:”

```
interface Initial_Interface;
typedef Initial_Interface Alias_Interface;
```

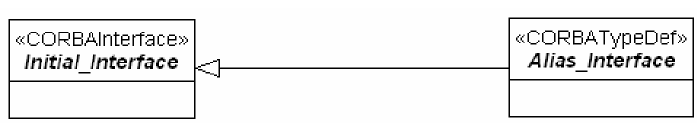

Figure 8.19 - TypeDef example

The second purpose is to provide a type name for anonymous template types, such as sequences or arrays. These typedefs are modeled by DataTypes that are stereotyped as <<CORBASequence>> and <<CORBAArray>> (see above).

CORBABoxedValue

Boxed values are similar to typedefs: they provide a new name for an existing type, and change the parameter passing semantics to allow instances of the new type to be null. When boxing an existing type declaration, the boxed value specializes the existing DataTypes (using a UML Generalization relationship) with a new DataType being the specialization, giving the type a new name, and possible null value semantics, but no new features. So, a boxed value type is represented by a UML DataType stereotyped as <<CORBABoxedValue>>.

The IDL below is represented in Figure 8.20.

```
valuetype OptionalNameSeq sequence<string>;
valuetype OptionalStruct my_struct;
```

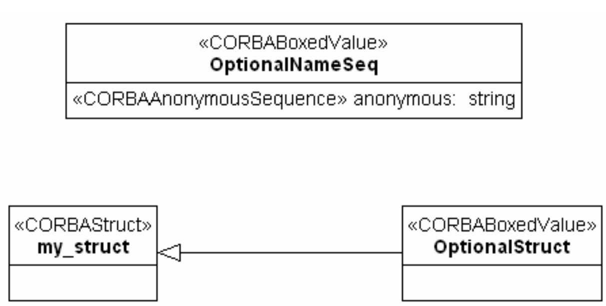

Figure 8.20 - BoxedValue example

8.1.3 Tabular Representation

Table 8.1 represents the BaseIDL Profile information.

Table 8.1 - BaseIDL Profile

Stereotype	Base Class	Parent	Tags	Constraints
CORBAInterface (InterfaceDef) <<CORBAInterface>>	Interface	N/A	isLocal: Boolean	[4]
CORBAValue (ValueDef) <<CORBAValue>>	Interface	N/A	isCustom: Boolean isTruncatable: Boolean	[5], [6], [7]
CORBAConstant (ConstantDef) <<CORBAConstant>>	Property	N/A		[14]
CORBAConstants <<CORBAConstants>>	Class	N/A		[12], [13]
CORBASupports (supportss) <<CORBASupports>>.	Generalization	N/A		[8]
CORBAValueFactory (ValueFactoryDef) <<CORBAValueFactory>>	Operation	N/A		[9], [10]
CORBAModule (ModuleDef) << CORBAModule>>	Package	N/A		[15]
CORBAPrimitive (PrimitiveDef) << CORBAPrimitive>>	DataType	N/A		[16]
CORBAConstructed <<CORBAConstructed>>	DataType	N/A		[17], [18]
CORBAUnion (UnionDef) <<CORBAUnion>>	DataType	CORBAConstructed		
CORBASwitch <<CORBASwitch>>	Property			
CORBAUnionField (UnionFieldDef) <<CORBAUnionField>>	Property			
CORBADefault <<CORBADefault>>	Property	CORBAUnionField		
CORBACase <<CORBACase>>	Property	CORBAUnionField	label: String	

Table 8.1 - BaseIDL Profile

CORBAStruct (StructDef) <<CORBAStruct>>	DataType	CORBAConstructed		[19]
CORBAException (ExceptionDef) <<CORBAException>>	DataType	CORBAConstructed		[20]
CORBAEnum (EnumDef) <<CORBAEnum>>	Enumeration			
CORBATemplate <<CORBATemplate>>	DataType		bound: Integer	
CORBAString (StringDef) <<CORBAString>>	DataType	CORBATemplate		
CORBAWstring (WstringDef) <<CORBAWstring>>	DataType	CORBATemplate		
CORBASequence (SequenceDef) <<CORBASequence>>	DataType	CORBATemplate		[21], [22]
CORBAAnonymousSequence (SequenceDef) <<CORBAAnonymousSequence>>	Property	CORBASequence		[23]
CORBAArray (ArrayDef) <<CORBAArray>>	DataType			[24]
CORBAAnonymousArray (ArrayDef) <<CORBAAnonymousArray>>	Property	CORBAArray	index:String	[25]
CORBAWrapper <<CORBAWrapper>>	DataType			[26] - [30]
CORBATypedef (TypedefDef) <<CORBATypedef>>	DataType	CORBAWrapper		[31]
CORBABoxedValue (ValueBoxDef) <<CORBABoxedValue>>	DataType	CORBAWrapper		

8.1.4 Constraints

[4] A <<CORBAInterface>>-stereotyped Interface tagged “isLocal” can only participate in Generalizations with other <<CORBAInterface>>-stereotyped Interfaces tagged “isLocal.”

context CORBAInterface inv:
(self.generalization->forAll(parent.isStereotyped("CORBAInterface") and
parent.stereotype.taggedValue->select(name = "isLocal")->size = 1)) and
(self.generalization->forAll(child.isStereotyped("CORBAInterface") and
child.stereotype.taggedValue->select(name = "isLocal")->size = 1))

[5] A concrete <<CORBAValue>>-stereotyped Interface may only specialize a single other concrete <<CORBAValue>>-stereotyped Interface.

context CORBAValue inv:
not self.isAbstract implies self.generalization->select(parent.isStereokinded("CORBAValue") and not parent.isAbstract)->size = 1

[6] A <<CORBAValue>>-stereotyped Interface may only specialize a single <<CORBAInterface>>-stereotyped Interface, and it must do so using a <<CORBAValueSupports>>-stereotyped Generalization.

context CORBAValue inv:
let supportedInterface = self.generalization->select(parent.isStereotyped ("CORBAInterface")) and
let supportsGeneralization = supportedInterface.generalization-> intersection(self.generalization) in
supportedInterface->size = 1 and supportsGeneralization.isStereotyped("CORBAValueSupports")

[7] A <<CORBAValue>>-stereotyped Class may only contain a single Operation stereotyped as <<CORBAValueFactory>>.

context CORBAValue inv:
self.allOperations->collect(isStereotyped("CORBAValueFactory"))->size <= 1

[8] A <<CORBASupports>>-stereotyped Generalization must have a <<CORBAInterface>>-stereotyped Interface as its parent and a <<CORBAValue>>-stereotyped Interface as its child.

context CORBASupports inv:
self.parent.isStereotyped("CORBAInterface") and self.child.isStereotyped("CORBAValue")

[9] A <<CORBAValueFactory>>-stereotyped Operation can have only in parameters and has no return type.

context CORBAValueFactory inv:
self.parameter->forAll(kind = #in)

[10] A <<CORBAValueFactory>>-stereotypedOperation must be owned by a <<CORBAValue>>-stereotyped or <<CORBACustomValue>>-stereotyped Class.

context CORBAValueFactory inv:
self.owner.isStereokinded("CORBAValue")

[11] A <<CORBAConstants>>-stereotyped Class must be directly contained by a <<CORBAModule>>-stereotyped package.

context CORBAConstant inv:
self.namespace.isStereotyped("CORBAModule")

[12] All the features of a <<CORBAConstants>>-stereotyped Class must be <<CORBAConstant>>-stereotyped Attributes.

context CORBAConstant inv:
self.feature->forAll(feature | feature.oclIsTypeOf (Property) and feature.isStereotyped ("CORBAConstant"))

[13] A <<CORBAConstants>>-stereotyped Class cannot participate in any Associations.

context CORBAConstant inv:
self.associations->isEmpty

[14] The owner of a <<CORBAConstant>>-stereotyped Property must be stereotyped <<CORBAConstants>>, <<CORBAInterface>> or <<CORBAValue>>.

context CORBAConstant inv:
self.owner.isStereotyped("CORBAConstants") or
self.owner.isStereokinded("CORBAInterface") or
self.owner.isStereokinded("CORBAValue")

[15] A <<CORBAModule>>-stereotyped package may directly contain at most one Class stereotyped as <<CORBAConstants>>.

context CORBAModule inv:
self.ownedElement->collect(el | el.isStereotyped("CORBAConstants"))->size <= 1

[16] All basic types (<<CORBAPrimitive>>-stereotyped UML DataTypes) are included in the package "CORBA.". The CORBA package also contains an Interface "Object," stereotyped as <<CORBAInterface>>, and an Interface "ValueBase," stereotyped as <<CORBAValue>>.

[17] All features of a <<CORBAConstructed>>-stereotyped Classifier must be Attributes with visibility "public." context CORBAConstructed inv:

self.feature->forAll(feature | feature.oclIsTypeOf(Attribute) and feature.visibility = #public)

[18] A <<CORBAConstructed>>-stereotyped Classifier cannot participate in any Generalization relationships.

context CORBAConstructed inv:
self.generalization->isEmpty and self.specialization->isEmpty

[19] All the Attributes of a <<CORBAStruct>>-stereotyped Classifier must have multiplicity 1..1.

context CORBAStruct inv:
self.allAttributes->forAll(multiplicity.range.lower = 1 and multiplicity.range.upper = 1)

[20] A <<CORBAException>>-stereotyped Exception cannot be the type of a navigable AssociationEnd.

context CORBAExeption inv:
self.allEnds->forAll(end | end.type = self implies not end.isNavigable)

[21] The single navigable opposite AssociationEnd of a <<CORBASequence>>- stereotyped Classifier must have multiplicity 1..1 if it cannot be a null in CORBA; that is, unless it is an object type or a boxed value type.

[22] The single navigable opposite AssociationEnd of a <<CORBASequence>>-stereotyped Classifier must have multiplicity 0..1 if it is a boxed value type or object type.

[23] A <<CORBAAnonymousSequence>>-stereotyped Class must have exactly one navigable opposite AssociationEnd whose multiplicity is 1..1.

context CORBAAnonymousSequence inv:
navigableOppositeEnds->size = 1 and navigableOppositeEnds ->forAll
(end | end.multiplicity.range.lower = 1 and end.multiplicity.range.upper = 1)

[24] The single navigable opposite AssociationEnd of a <<CORBAArray>>- stereotyped Class must have multiplicity 1..1.

context CORBAArray inv:
navigableOppositeEnds->forAll
(end | end.multiplicity.range.lower = 1 and end.multiplicity.range.upper = 1)

[25] A <<CORBAAnonymousArray>>-stereotyped Class must have exactly one navigable opposite AssociationEnd whose multiplicity is 1..1.

context CORBAAnonymousArray inv:
navigableOppositeEnds->size = 1 and navigableOppositeEnds->forAll
(end | end.multiplicity.range.lower = 1 and end.multiplicity.range.upper = 1)

[26] A <<CORBAWrapper>>-stereotyped Classifier must participate as the child in exactly one Generalization relationship.

context CORBAWrapper inv:
self.generalization->select(gen | gen.child = self)->size = 1

[27] The Generalization relationship in which a <<CORBAWrapper>>-stereotyped Classifier participates has the empty string as its discriminator and no powertypes.

context CORBAWrapper inv:
self.generalization->forAll(gen | gen.discriminator = "" and
gen.powertype->isEmpty)

[28] A <<CORBAWrapper>>-stereotyped Classifier may not have any non-inherited features.

context CORBAWrapper inv:
self.feature->isEmpty

[29] A <<CORBAWrapper>>-stereotyped Classifier may not participate in any Associations with navigable opposite AssociationEnds.

context CORBAWrapper inv:
self.navigableOppositeEnds->isEmpty

[30] A <<CORBAWrapper>> can only extend a DataType or a Interface.

context CORBAWrapper inv:
self.oclIsTypeOf(DataType) or self.oclIsTypeOf(Interface)

[31] The parent of a <<CORBATypedef>>-stereotyped Classifier must not be stereotyped as <<CORBAAnonymousSequence>> or <<CORBAAnonymousArray>>.

context CORBATypedef inv:

self.generalization->forAll (gen |not gen.parent.isStereotyped("CORBAAnonymousSequence") and
not gen.parent.isStereotyped("CORBAAnonymousArray"))

8.2 ComponentIDL Profile

8.2.1 Stereotypes

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 represented by the <<CORBASupports>> stereotyped generalization defined in BaseIDL Profile.

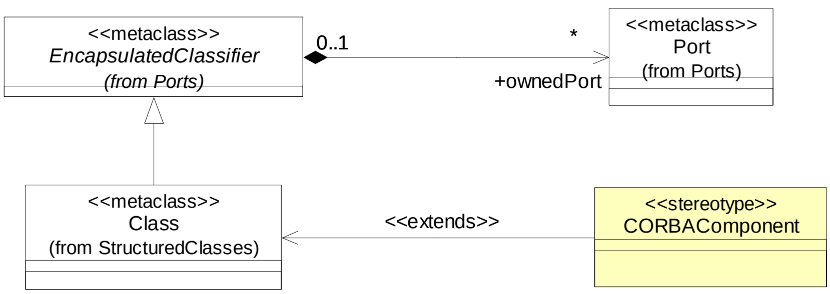

Figure 8.21 - ComponentIDL Profile: Extended UML classes (I)

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 different kinds of ports: facets, receptacles, event ports, and stream ports.

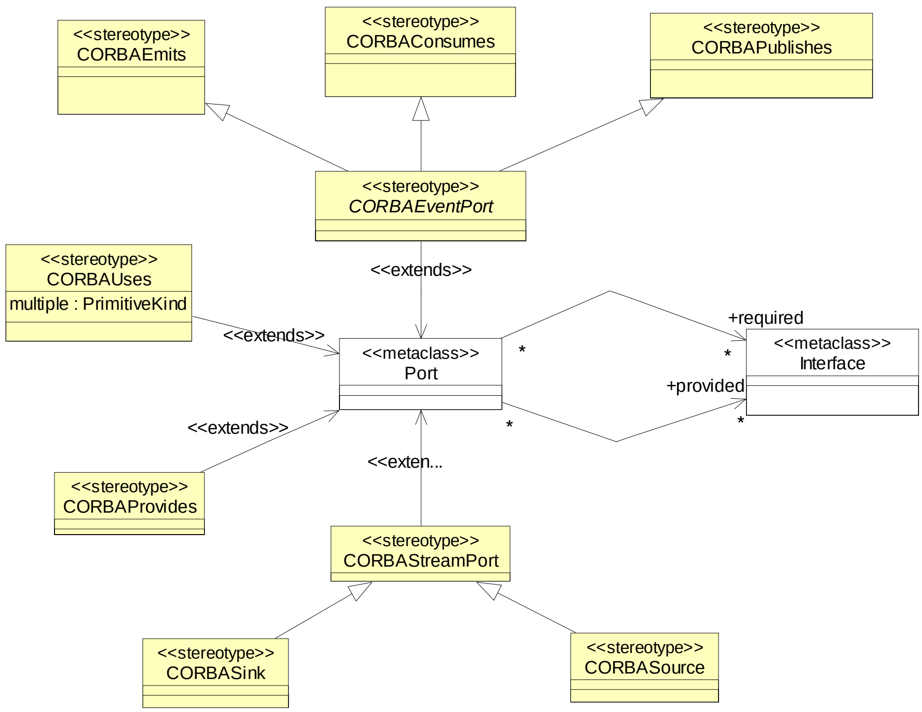

Figure 8.22 ComponentIDL Profile: Extended UML classes (II)

The facet definitions are represented by a UML Port stereotyped as <<CORBAProvides>>.

The receptacle definitions are represented by a UML Port stereotyped as <<CORBAUses>>. The tag “multiple” by the <<CORBAUses>> indicates whether the multiple connections to the receptacle may exist simultaneously or not.

The component has event ports. 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. The extended UML metamodel contains an abstract stereotype <<CORBAEventPort>>, which extends a UML Port and generalizes <<CORBAConsumes>> stereotype representing port where events are consumed, <<CORBAEmits>> stereotype where events are published only to one consumer, and <<CORBAPublishes>> stereotype where events are published to several consumers.

For the stream communication components have stream ports. The abstract stereotype <<CORBAStreamPort>> represents stream ports and generalizes stereotypes <<CORBASource>> for a source port and <<CORBASink>> for sink ports. A stream type is represented by a UML Interface stereotyped as <<CORBAStream>>. The tag “*kind*” identifies the kind of the <<CORBAStream>>.

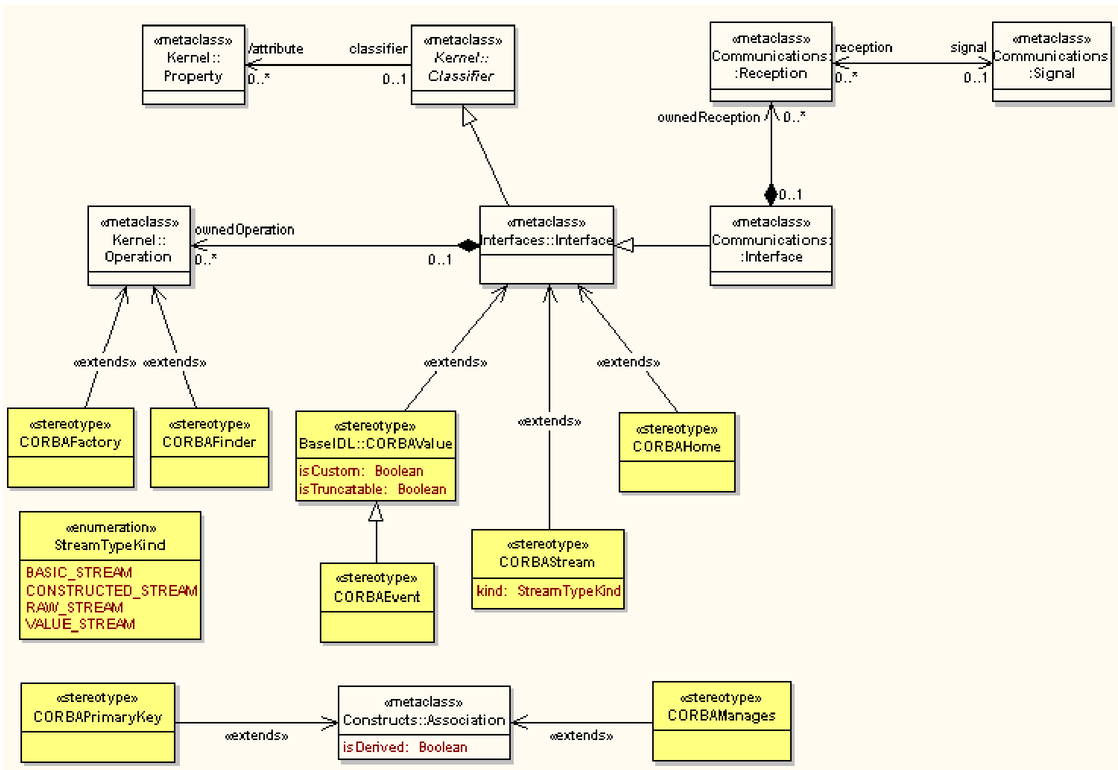

Figure 8.23 - ComponentIDL Profile: Extended UML classes (III)

A Component home is represented by a UML Interface stereotyped as <<CORBAHome>>. A component home must be associated to a component type. This relationship is made explicit using a <<CORBAManages>> stereotyped UML Association between a <<CORBAHome>> and a <<CORBAComponent>>. <<CORBAHome>> can inherit from another <<CORBAHome>> (single inheritance) using a UML Generalization. <<CORBAHome>> can support several <<CORBAInterface>>, this relationship is represented by the stereotyped as <<CORBASupports>> Generalization.

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 UML Interface, the relationship between home and its primary key is represented by an Association stereotyped as <<CORBAPrimaryKey>>.

<<CORBAHome>> can own attributes and operations. A UML Operation stereotyped as <<CORBAFactory>> is used to represent component factory operations, and as <<CORBAFinder>> is used to represent components finder operations.

Event types are represented by a UML Interface stereotyped as <<CORBAEvent>>. The <<CORBAEvent>> stereotype is a specialization of the <<CORBAValue>> stereotype. It inherits from all <<CORBAValue>> constraints.

8.2.2 Tabular Representation

Table 8.2 represents the ComponentIDL Profile information.

Table 8.2 - ComponentIDL Profile

Stereotype	Base Class	Parent	Tags	Constraints
CORBAComponent (ComponentDef) <<CORBAComponent>>	Class			[32] - [35]
CORBAProvides (ProvidesDef) <<CORBAProvides>>	Port			
CORBAUses (UsesDef) <<CORBAUses>>	Port		isMultiple: Boolean	
CORBAEventPort (EventPortDef) <<CORBAEventPort>>	Port			
CORBAEvent (EventDef) <<CORBAEvent>>	Interface			
CORBAEmits (EmitsDef) <<CORBAEmits>>	Port	CORBAEventPort		
CORBAPublishes (PublishesDef) <<CORBAPublishes>>	Port	CORBAEventPort		
CORBAConsumes (ConsumesDef) <<CORBAConsumes>>	Port	CORBAEventPort		
CORBAStream (StreamTypeDef) <<CORBAStream>>	Interface		kind: StreamKind	
CORBAStreamPort (StreamPortDef) <<CORBAStreamPort>>	Port			
CORBASource (SourceDef) <<CORBASource>>	Port	CORBAStreamPort		
CORBASink (SinkDef) <<CORBASink>>	Port	CORBAStreamPort		
CORBAHome (HomeDef) << CORBAHome >>	Interface			[39] - [42]
CORBAFactory (FactoryDef) << CORBAFactory >>	Operation			[44], [45]
CORBAFinder (FinderDef) << CORBAFinder >>	Operation			[46], [47]

Table 8.2 - ComponentIDL Profile

CORBAManages << CORBAManages >>	Association			[36] - [38]
CORBAPrimaryKey << CORBAPrimaryKey>>	Association			
CORBAValue (ValueDef) <<CORBAValue>>	Interface			[43]

Constraints

[32] A "CORBAComponent" cannot own operations.

context CORBAComponent inv:
self.feature forAll(not oclIsKindOf (behavioralFeature))

[33] A "CORBAComponent" can only inherit from a "CORBAComponent" or a "CORBAInterface"

context CORBAComponent inv:
self.generalization -> forAll (g : Generalization | g.parent.isStereotyped ("CORBAComponent") or g.parent.isStereotyped("CORBAInterface"))

[34] Only single inheritance is possible between "CORBAComponent."

context CORBAComponent inv:
self.generalization -> select(parent.isStereotyped("CORBAComponent")) ?size <= 1

[35] Each "CORBAComponent" inheritance from a "CORBAInterface" must be stereotyped "CORBASupports."

context CORBAComponent inv:
self.generalization -> forAll (g : Generalization | g.parent.isStereotyped("CORBAInterface") implies g.isStereotyped("CORBASupports"))

[36] There is exactly one "CORBAManages" association for each Home.

context CORBAManages inv:
self.connection ?select(isStereotyped("CORBAManages")) ->size = 1

[37] The "CORBAHome" side cardinality must be 1..1.

context CORBAHome inv:
self.connection ?exists(participant.isStereotyped("CORBAHome")) and multiplicity.min=1 and multiplicity.max=1)

[38] The "CORBAComponent" side cardinality must be "0..n."

context CORBAComponent inv:
self.connection ?exists(participant.isStereotyped("CORBAComponent")) and multiplicity.min=0 and multiplicity.max=n)

[39] A "CORBAHome" can inherit from one "CORBAHome" at most.

context CORBAHome inv:
self.generalization ?select(parent.isStereotyped("CORBAHome")) ?size=1

[40] If “CORBAHome” h1 inherits from “CORBAHome” h2 and h2 manages “CORBAComponent” C2 then h1 must manage C2 or any other component C1 that inherits from C2.

context CORBAHome inv:
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))

[41] 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.

context CORBAHome inv:
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))

[42] Each “CORBAHome” inheritance from a “CORBAInterface” must be stereotyped.

context CORBAHome inv:
self.generalization ->forAll g,Generalization | g.parent.isStereotyped ("CORBAInterface")
implies g.isStereotyped("CORBASupports"))

[43] The valuetype of a primary key:

[43-1] must not have private state members

[43-2] must not have members that are interfaces

[43-3] must have at least one state member

[43-5] must descend directly or indirectly from Components::PrimaryKeyBase

[43-4] Contraints [43-1], [43-2], and [43-3] apply recursively to valuetype members that are valuetypes.

[43-1, 43-2, 43-3, 43-4] 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))

}

[43-5] 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 }
```

[44] A "CORBAHomeFactory" operation has only input parameters.

```
context CORBAHomeFactory inv:
        self. parameter ?forAll(kind=#in)
```

[45] A "CORBAHomeFactory" can only be defined in a "CORBAHome."

```
context CORBAHomeFactory inv:
        self.owner.isStereotyped("CORBAHome")
```

[46] A "CORBAHomeFinder" operation has only input parameters.

```
context CORBAHomeFinder inv:
        self. parameter ?forAll(kind=#in)
```

[47] A "CORBAHomeFinder" can only be defined in a "CORBAHome."

```
context CORBAHomeFinder inv:
        self.owner.isStereotyped("CORBAHome")
```

8.2.3 Example

Following IDL describes the Philosophers example:

```
typedef enum PhilosopherState

{

        EATING,
        THINKING,
        HUNGRY,
        STARVING,
        DEAD};

eventtype StatusInfo {

        public string  name;
        public PhilosopherState state;
        public long secondesSinceLastMeal;
        public boolean hasLeftFork;
        public boolean hasRightFork;};


exception InUse {};

// Interfaces

interface Registration {
```

```
	string register();};

interface Fork
{
	void get() raises (InUse);
	void release();};

//Components and Homes

component Philosopher {
	uses Fork left;
	uses Fork right;
	uses Registration registration;
	publishes StatusInfo info;};

home PhilosopherHome manages Philosopher {};


component Fork {
	provides Fork the_fork;};

home ForkHome manages Fork {};


component Registrator supports Registration {};

home RegistratorHome manages Registrator {};


component Observer {
	consumes StatusInfo info;};

home ObserverHome manages Observer {};
```

The UML model of components described in IDL above is shown in the figure below:

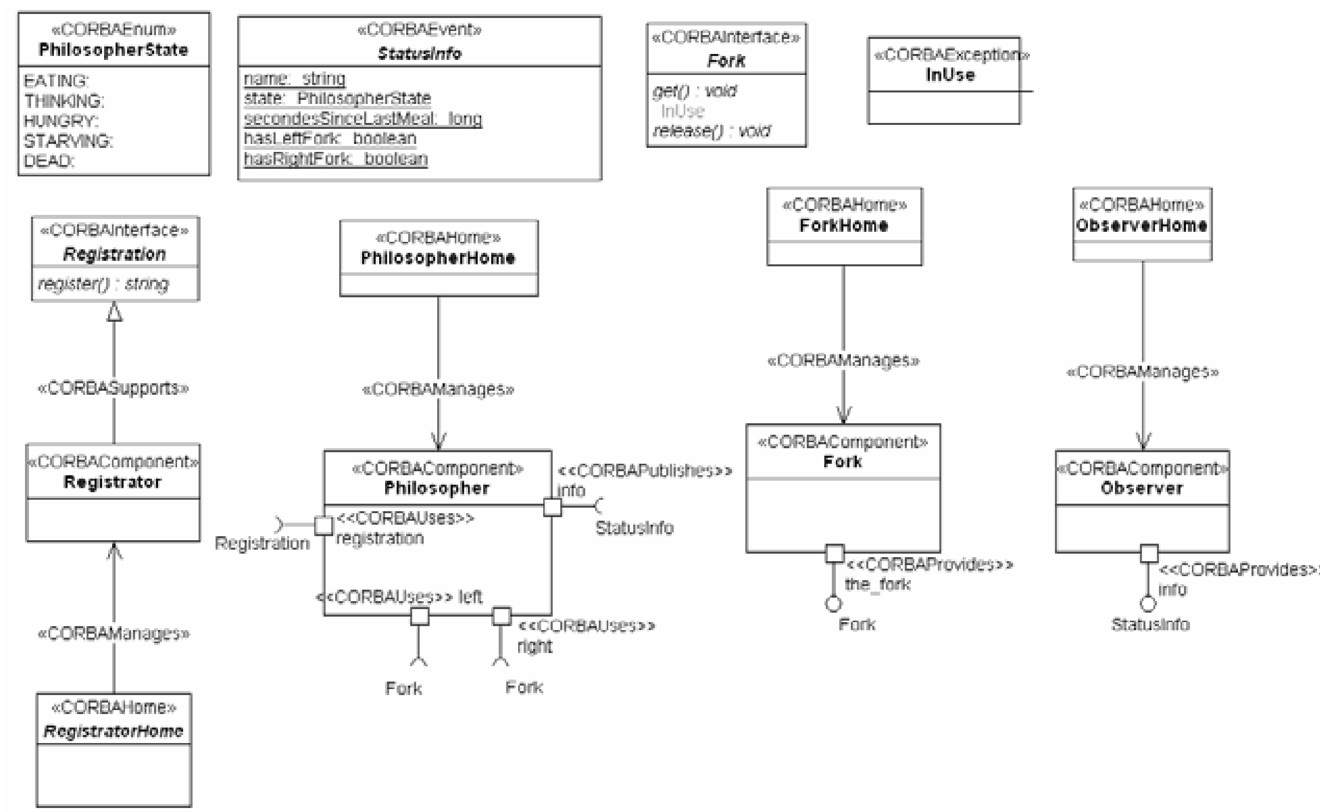

Figure 8.24 - ComponentIDL Profile example

8.3 CIF Profile

8.3.1 Stereotypes

The CIF Profile defines how CORBA components have to be implemented. An implementation of a component comprises a potentially complex set of artifacts (e.g., component or home executors) that must exhibit specific relationships and behaviors in order to provide a proper implementation. A composition is a unit of component implementation and contains such artifacts. A composition is represented using a UML Component (from the package “PackagingComponents”) with the stereotype <<CORBAComposition>>.

A component implementation is represented using a UML Class with the stereotype <<CORBAComponentExecutor>>. The <<CORBAComponentExecutor>> is always defined within a <<CORBAComposition>> element.

A home implementation is represented using a UML Class with the stereotype <<CORBAHomeExecutor>>. The <<CORBAHomeExecutor>> is always defined within a <<CORBAComposition>> element.

The relationships between components and component executors and between homes and home executors are represented by an Association stereotyped as <<CORBAImplements>>.

A segment is represented using a UML Part (Property) with the stereotype <<CORBASegment>>. Segments are physical partitions of a <<CORBAComponentExecutor>> element and always represented in the internal structure of a <<CORBAComponentExecutor>> element (as UML parts, see example in Figure 8.26).

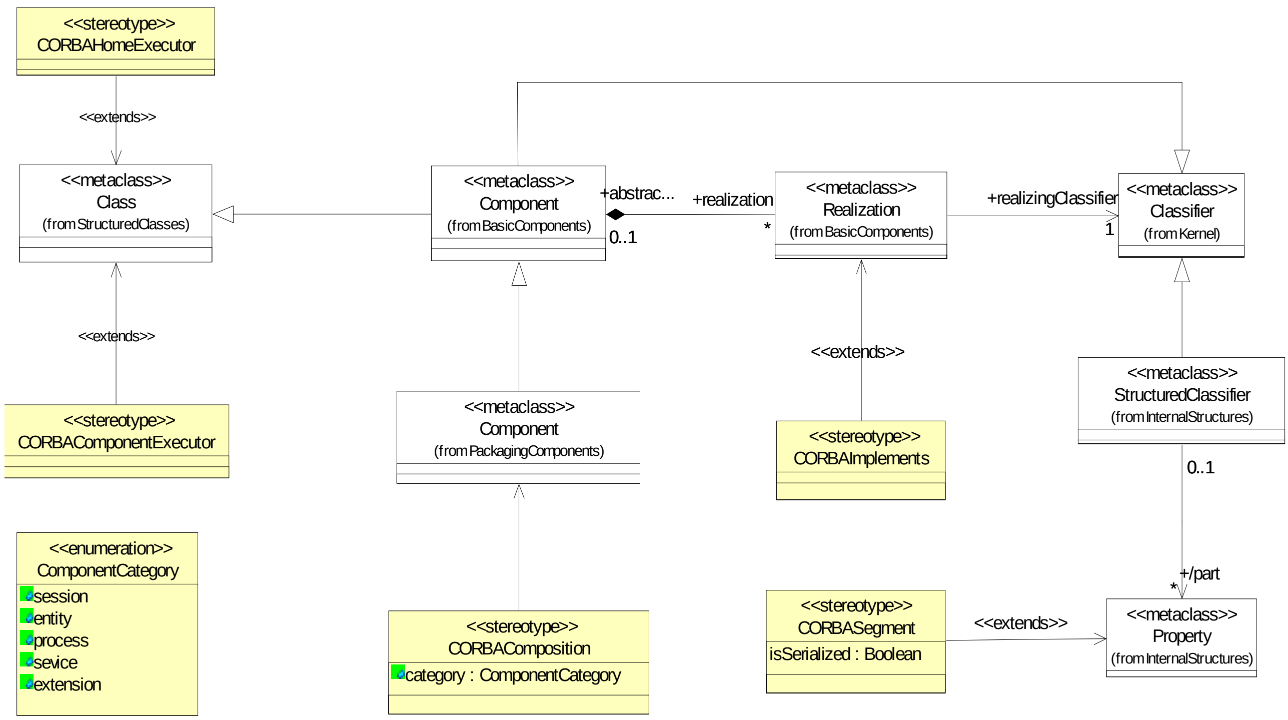

Figure 8.25 - CIF Profile: Extended UML classes

8.3.2 Tabular Representation

Table 8.3 represents the CIF Profile information.

Table 8.3 - CIF Profile

Stereotype	Base Class	Parent	Tags	Constraints
CORBAComponentExecutor (ComponentExecutorDef) <<CORBAComponentExecutor>>	Class			[49]
CORBAHomeExecutor (HomeExecutorDef) <<CORBAHomeExecutor>>	Class			[53]
CORBAImplements <<CORBAImplements>>	Realization (Association)			[48], [50]

Table 8.3 - CIF Profile

CORBAManages << CORBAManages>> (from ComponentIDL Profile)	Association			[51], [52]
CORBAComposition (CompositionDef) <<CORBAComposition>>	Component		category: ComponentCategory	[54]
CORBASegment (SegmentDef) <<CORBASegment>>	Property (Part)		isSerialized: Boolean	[55]

8.3.3 Constraints

[48] There is an association between <<CORBAComponentExecutor>> and <<CORBAComponent>>.

context CORBAImplements inv:
self.connection ? exists(participant.isStereotyped("CORBAComponentExecutor")) and
self.connection ?exists(participant.isStereotyped("CORBAComponent"))

[49] A <<CORBAComponentExecutor>> always has exactly one <<CORBAComponent>> associated while each <<CORBAComponent>> might be implemented by different types of <<CORBAComponentExecutor>>.

context CORBAComponentExecutor inv:
self.connection ? exists(participant.isStereotyped("CORBAComponentExecutor") and multiplicity.min=1 and max=*)
self.connection ?exists(participant.isStereotyped("CORBAComponent") and multiplicity.min=1 and max=1)

[50] Each <<CORBAHomeExecutor>> in a model implements exactly one <<CORBAHome>>.

context CORBAHomeExecutor inv:
self.connection ?exists(participant.isStereotyped("CORBAHomeExecutor") and multiplicity.min=1 and max=1)
self.connection ?exists(participant.isStereotyped("CORBAHome") and multiplicity.min=1 and max=1)

[51] It's an association between a "CORBAHomeExecutor" and a "CORBAComponentExecutor."

context CORBAManages inv:
self.connection ?exists(participant.isStereotyped("CORBAHomeExecutor")) and
self.connection ? exists(participant.isStereotyped("CORBAComponentExecutor"))

[52] Each <<CORBAHomeExecutor>> manages exactly one <<CORBAComponentExecutor>>, this relation is modeled by the association <<CORBAManages>>.

context CORBAHomeExecutor inv:
self.connection ? exists(participant.isStereotyped("CORBAComponentExecutor") and multiplicity.min=1 and max=1)

[53] For each instance x of <<CORBAHomeExecutor>> 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 <<CORBAComponentExecutor>>, which is associated to x.

context CORBAHomeExecutor inv:
self.home.component = self.component_impl.component

[54] The life cycle category of the <<CORBAComposition>> must be “entity” or “process” if the contained component implementation is segmented.

context CORBAComposition inv:
self.component_impl.segments>1 implies (self.category=ENTITY or self.category=PROCESS)

[55] <<CORBASegment>> classes are always contained in <<CORBAComponentExecutor>>.

context CORBASegment inv:
self.definedIn.oclIsTypeOf(ComponentExecutorDef)

8.3.4 Example

The following IDL for the component Fork from the Philosophers example is represented in Figure 8.26:

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

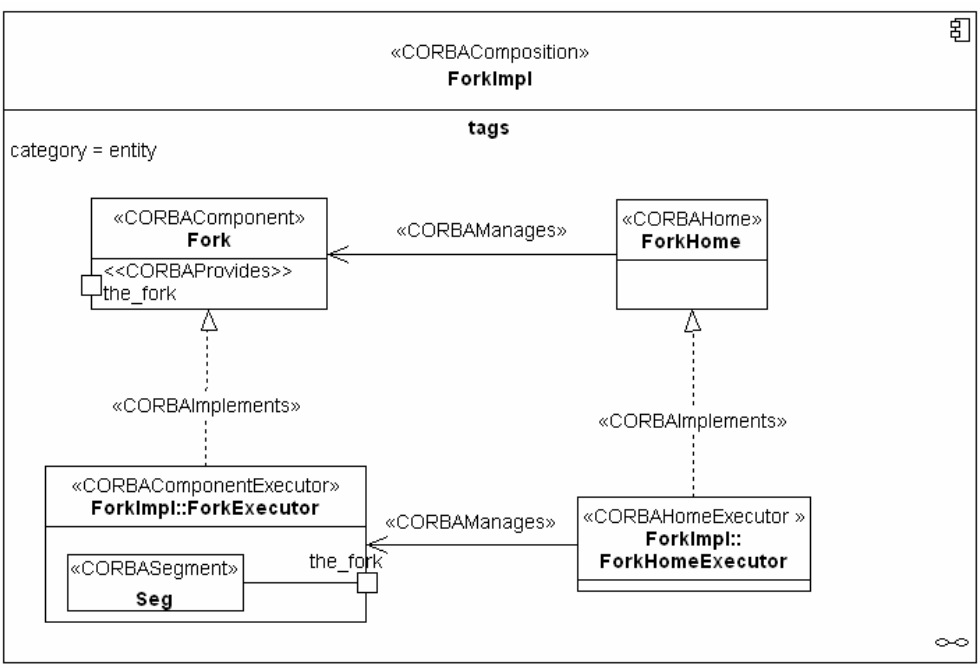

Figure 8.26 - CIF Profile example

8.4 Deployment Profile

The Component Implementation Framework (CIF) Profile defines how to model constructing component implementations. How to model components and component homes is defined in the ComponentIDL Profile (see previous sections). The CIF Profile uses ComponentIDL Profile descriptions to model CCM applications and then generate programming skeletons that automate many of the basic behaviors of components, including navigation, identity inquiries, activation, state management, lifecycle management, and so on. Generated CCM components are units of deployment process, which includes installation, configuration, planning, preparation, and launch of such CCM applications. In order to deploy a component-based application like CCM applications instances of each component must first be created, then interconnected and configured. The Deployment Profile defines how to model deployment and configuration information of CCM applications. The Deployment Profile uses the CIF Profile (e.g., for modeling of component and home executors) and introduces possibilities for modeling of an initial configuration: a set of interconnected component instances (assembly) of a CCM application at run time and other deployment information.

8.4.1 Stereotypes

A CORBA Assembly package is represented using a UML Package with the stereotype <<CORBAAssemblyPkg>>. The <<CORBAAssemblyPkg>> element may contain one or more component packages represented using a UML Package with the stereotype <<CORBAAComponentPkg>>.

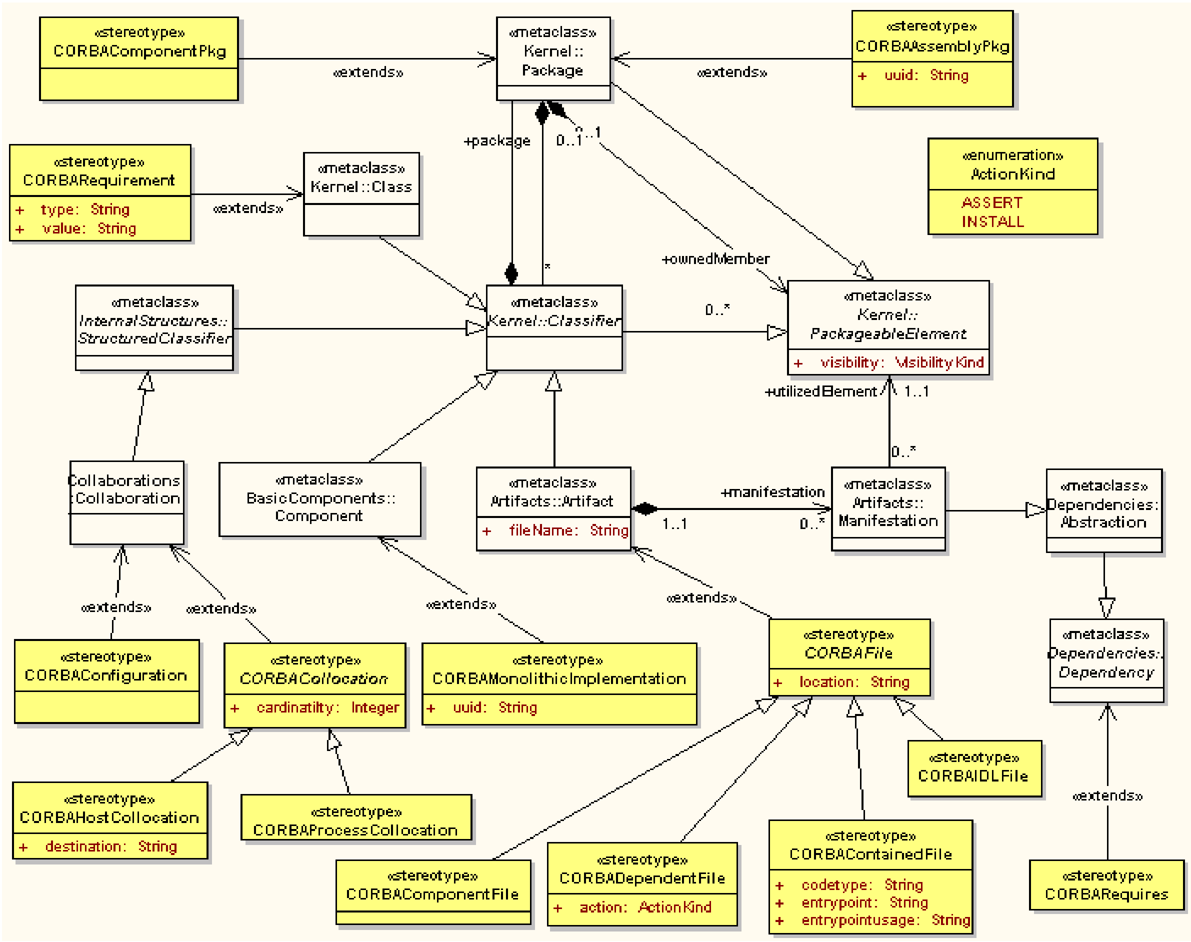

Figure 8.27 - Deployment Profile: Extended UML classes

A component package <<CORBAAComponentPkg>> is a set of metadata (e.g., IDL description) and compiled code modules that contain implementations of a component. The implementations in <<CORBAAComponentPkg>> can be monolithic and represented as a UML Component with the stereotype <<CORBAMonolithicImplementation>> or in the form of an assembly and represented as a UML Package with the stereotype <<CORBAAssemblyPkg>>. <<CORBAMonolithicImplementation>> elements can be described by platform dependencies, code filename, entry points, and other deployment characteristics. These characteristics or implementation requirements can be represented by the stereotyped UML class <<CORBARequirement>>. The possible requirements of an implementation are listed below:

- licensekey: point to the key of a usage license
- licensetextref: point to the text of a usage license
- uuid: unique identifier of an implementation
- compiler: specifies the compiler used to create an implementation
- programminglanguage: specifies the type of the component implementation
- description: string description for any additional information
- humanlanguage: specifies a spoken language

- os: specifies a particular operating system that the implementation will work with
- processor: indicates the type of processor that the implementation must run on

The name of the <<CORBARequirement>> class instance can be one of the names listed above (or other), the tag "*type*" of the <<CORBARequirement>> class represents a concrete type of the requirement and the tag "*value*" of the class helps to define a concrete version of the requirement's type. For example, the UML class "Processor" with the stereotype <<CORBARequirement>> can have tagged values "type=Intel" and "value=Core™2 Duo."

The initial configuration of a CCM application is represented as a UML Collaboration with the stereotype <<CORBAConfiguration>> and contains instances of component and home implementations. These instances can be collocated in the same process or run on the same node (host). For these two kinds of collocation representation <<CORBAProcessCollocation>> and <<CORBAHostCollocation>> stereotypes are defined. They both inherit from the abstract stereotype class <<CORBACollocation>>, which have the "cardinality" tag. The "cardinality" tag represents how many instances of collocation may be deployed.

The assembly and component packages may contain different deployment artifacts: specifications of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of such artifacts are all defined in the Deployment metamodel files: component, IDL, contained and dependent files described in Section 3.2.4. All these files are represented using a UML Artifact with stereotypes <<CORBAAComponentFile>>, <<CORBAAIDLFile>>, <<CORBAAContainedFile>>, and <<CORBAADependentFile>>. These files are usually required from implementations. The relationship between a <<CORBAImplementation>> element and required artifacts is represented using the UML Dependency with a stereotype <<CORBARequires>>.

8.4.2 Tabular Representation

Table 8.4 represents the Deployment Profile information.

Table 8.4 - Deployment Profile

Stereotype	Base Class	Parent	Tags	Const-raints
CORBARequirement (RequirementDef) <<CORBARequirement>>	Class		type: String value: String	
CORBAFile (File) <<CORBAFile>>	Artifact			
CORBAMonolithicImplementation (MonolithicImplementationDef) <<CORBAImplementation>>	Component			
CORBAContainedFile (ContainedFile) <<CORBAContainedFile>>	Artifact	CORBAFile	codetype: String, entrypoint: String, entrypointusage: String	
CORBADependentFile (DependentFile) <<CORBADependentFile>>	Artifact	CORBAFile	action: ActionKind	
CORBAComponentFile (ComponentFile) <<CORBAComponentFile>>	Artifact	CORBAFile		

Table 8.4 - Deployment Profile

CORBAIDLFile (IDLFile) <<CORBAIDLFile>>	Artifact	CORBAFile		
CORBARequires <<CORBARequires>>	Dependency			
CORBAAssemblyPkg (AssemblyPkgDef) <<CORBAAssembly>>	Package			
CORBAComponentPkg (ComponentPkgDef) <<CORBAComponentPkg>>	Package			
CORBAProcessCollocation (ProcessCollocationDef) <<CORBAProcessCollocation>>	Collaboration	CORBA Collocation		
CORBAHostCollocation (HostCollocationDef) <<CORBAHostCollocation>>	Collaboration	CORBA Collocation	destination:String	
CORBACollocation (CollocationDef) <<CORBACollocation>>	Collaboration		cardinality:String	
CORBAConfiguration (ConfigurationDef) <<CORBAConfiguration>>	Collaboration			

8.4.3 Constraints

There are no specific constraints for this profile.

8.4.4 Example

There is no IDL notation for deployment information: as input to a deployment tool, component and assembly packages are provided. These packages contain one or more XML descriptors and a set of files. The XML descriptors contain all needed deployment and configuration data used by a deployment tool.

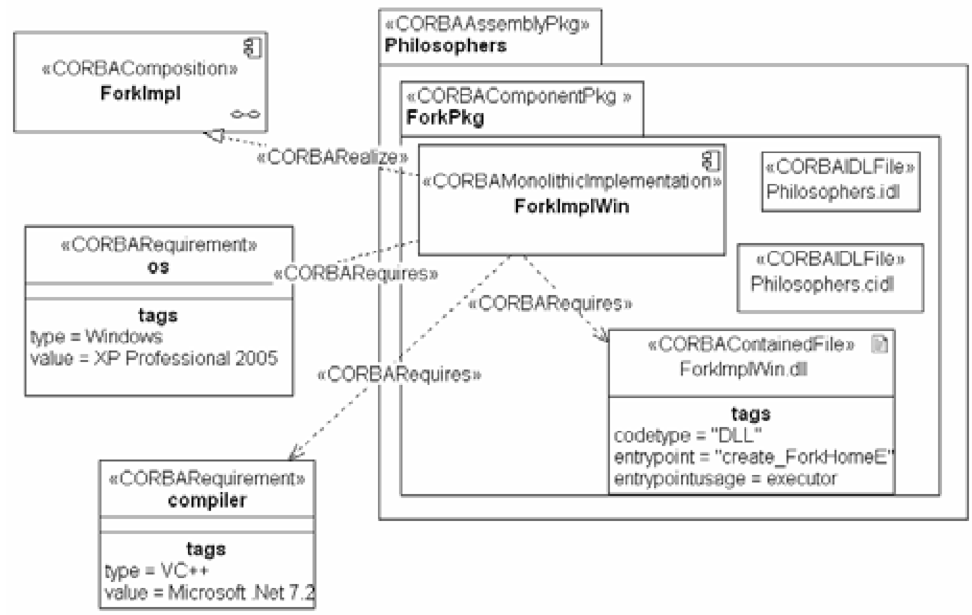

Figure 8.28 - Deployment Profile example: package structure of an application

The figure below represents one possible initial configuration of the Philosophers application:

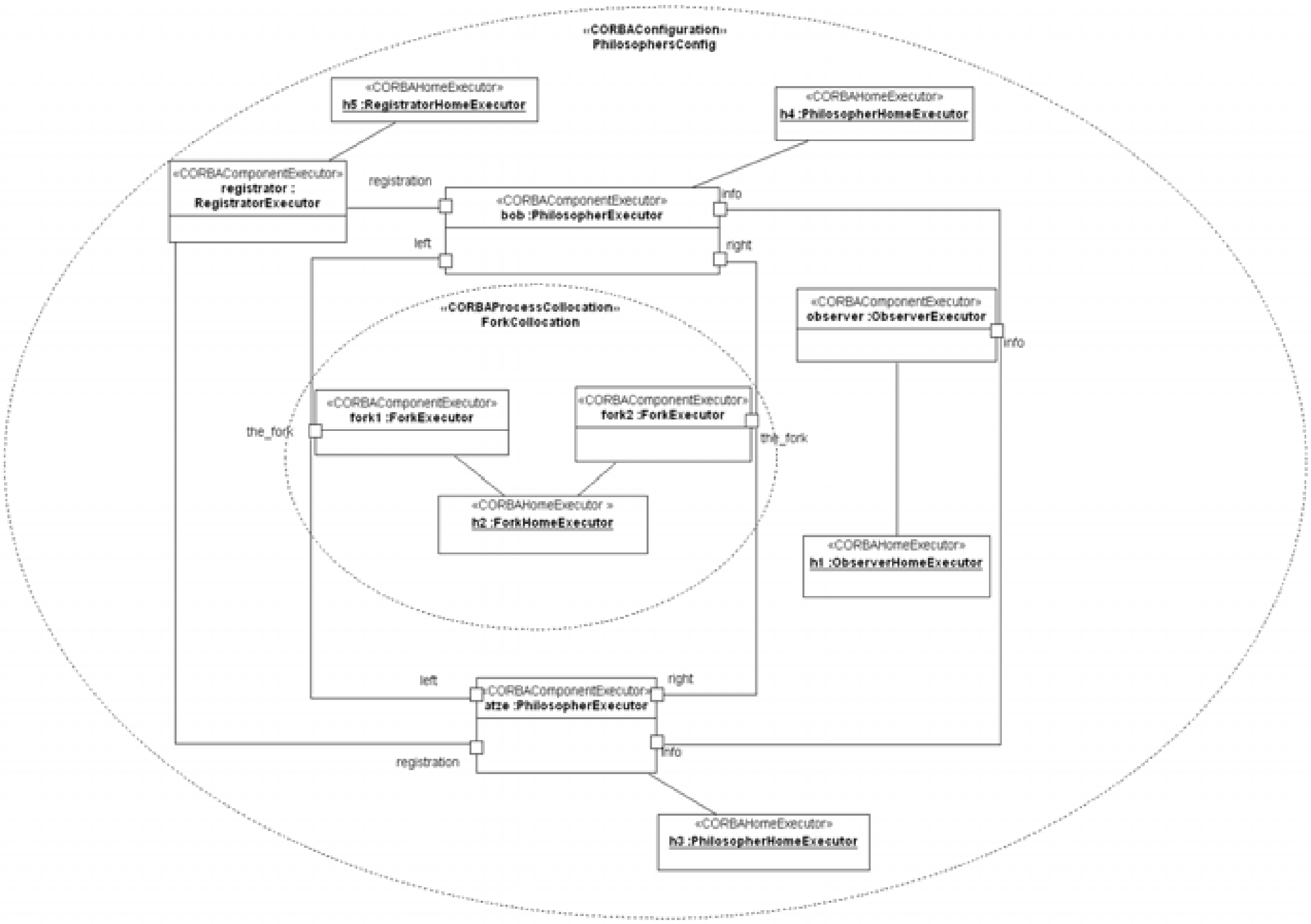

Figure 8.29 - Deployment Profile example: initial configuration of an application

8.5 CCMQoS Profile

The “UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms” (ptc/05-05-02) defines a comprehensive UML 2.0 profile for the description of QoS properties. The modeling of QoS properties for CORBA Components requires the definition of a link between QoS profile and CCM profile. This link is defined in the profile package CCMQoS and is a small extension done by the stereotype QoSBinding (see Figure 8.30). The QoSBinding class extends the UML metaclass Comment and can be attached to any UML element instance.

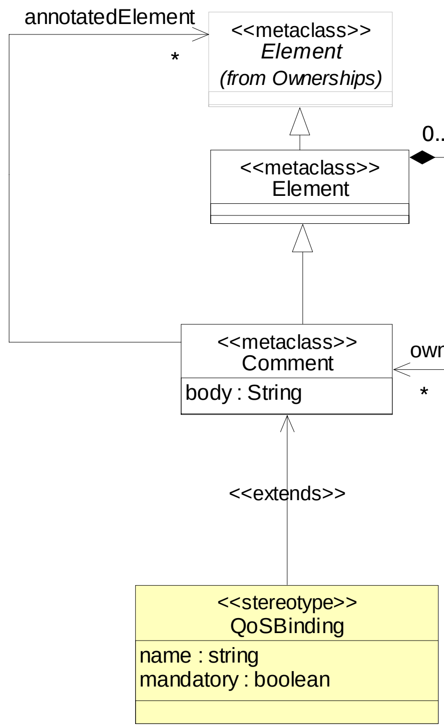

Figure 8.30 - CCMQoS Profile: extended UML classes

The example shown below describes a simple video service: stream-based communication between two CORBA components StreamClient and StreamServer. The stereotyped with <<QoS Characteristics>> VideoQoS class represents quantifiable characteristics (dimensions for the quantification) of the video services. For more information about QoS please refer to the OMG document ptc/05-05-02.

The Binding metaclass is represented using a UML Comment with stereotype <<QoSBinding>> and can be attached either to a component or its port or its instances. The <<QoSBinding>> element has two tags: "*name*" and "*mandatory.*"

8.5.1 Tabular Representation

Table 8.5 represents the CCMQoS Profile information.

Table 8.5 - CCMQoS Profile

Stereotype	Base Class	Parent	Tags	Constraints
QoSBinding (Binding) <<QoSBinding>>	Comment		name: string mandatory: boolean	

8.5.2 Constraints

There are no specific constraints for this profile.

8.5.3 Example

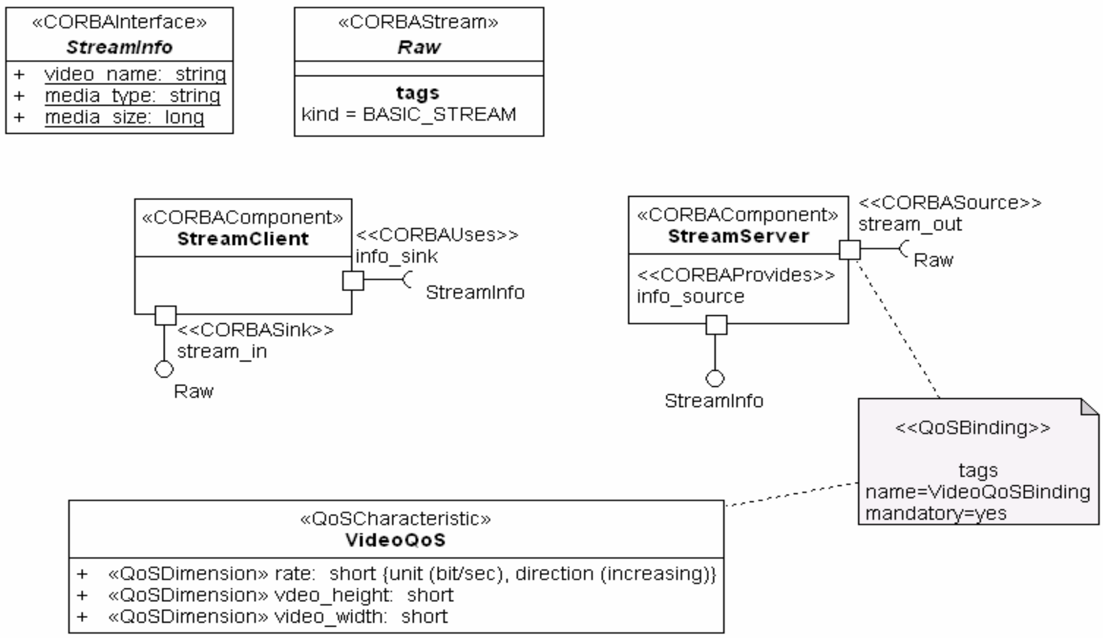

Figure 8.31 - CCMQoS Profile: video example

8.6 UML Profile for Lightweight CCM

This profile provides modeling concepts of the basic level of CORBA Components defined as Lightweight CCM Profile and specified in the CCM specification (formal/06-04-01).

The Lightweight CCM profile provides an enriched environment for low-footprint, embedded, and real-time CORBA solutions. It considers only specific parts of the CCM specification that are impacted and the normative specific subsetting of CCM. General CCM capabilities and support, such as Persistence, CIDL, Home finders, or Configuration are not included in Lightweight CCM (see formal/06-04-01, section 13).

The UML Profile for Lightweight CCM is defined as a subset of the CCM Profile described in previous sections. There are no new stereotypes or tags for this profile added. The following changes made in Lightweight CCM do not have a bearing on the UML Profile for Lightweight CCM:

- Changes associated with excluding support for introspection, navigation, and type-specific operations redundant with generic operations.
- Changes associated with excluding support for transactions.
- Changes associated with excluding support for security.
- Changes associated with excluding support for configurators.
- Changes associated with excluding support for proxy homes.

Changes associated with excluding support for persistence, segmentation and home finders impact CCM Metamodel and Profile. Following normative exclusions described in the Lightweight CCM profile have been taken into account:

Table 8.6 - Lightweight CCM

Normative Exclusion	**Metamodel impacts**	**Profile impacts**
Exclude support for primary keys.	The association “key_home” between metaclasses *HomeDef* and *ValueDef* has been removed.	The Stereotype <<CORBAPrimaryKey>> has been removed.
Exclude support for CIDL.	Excluding CIDL is a result of excluding support for both persistence and segmentation (see Exclusions below).	
Exclude composition	The metaclass *CompositionDef* has been removed. The attributes of the enumeration *ComponentCategory* “process”, “entity” and “extension” have been removed. The attribute “category” has been added to the metaclass *ComponentExcecutorDef*.	The Stereotype <<CORBAComposition>> has been removed. The tag “category” has been moved from the stereotype <<CORBAComposition>> to the <<CORBAComponentExecutor>>
Restrict CIF metamodel to a single segment per implementation	The multiplicity of the association between *ComponentExcecutorDef* and *SegmentDef* has been changed to 1:1.	Following Constraint has been added: ***[56]*** ***context CORBAComponentExecutor inv:*** ***self.connection ? exists*** ***(participant.isStereotyped ("CORBASegment")*** ***and multiplicity.min=1 and max=1)***
Remove segmentation support	(see previous Exclusion)	(see previous Exclusion)
Exclude support for home finders and finder operations	The *FinderDef* metaclass has been removed.	The Stereotype <<CORBAFinder>> has been removed

The following figures represent changed metamodels (ComponentIDL and CIF) for Lightweight CCM:

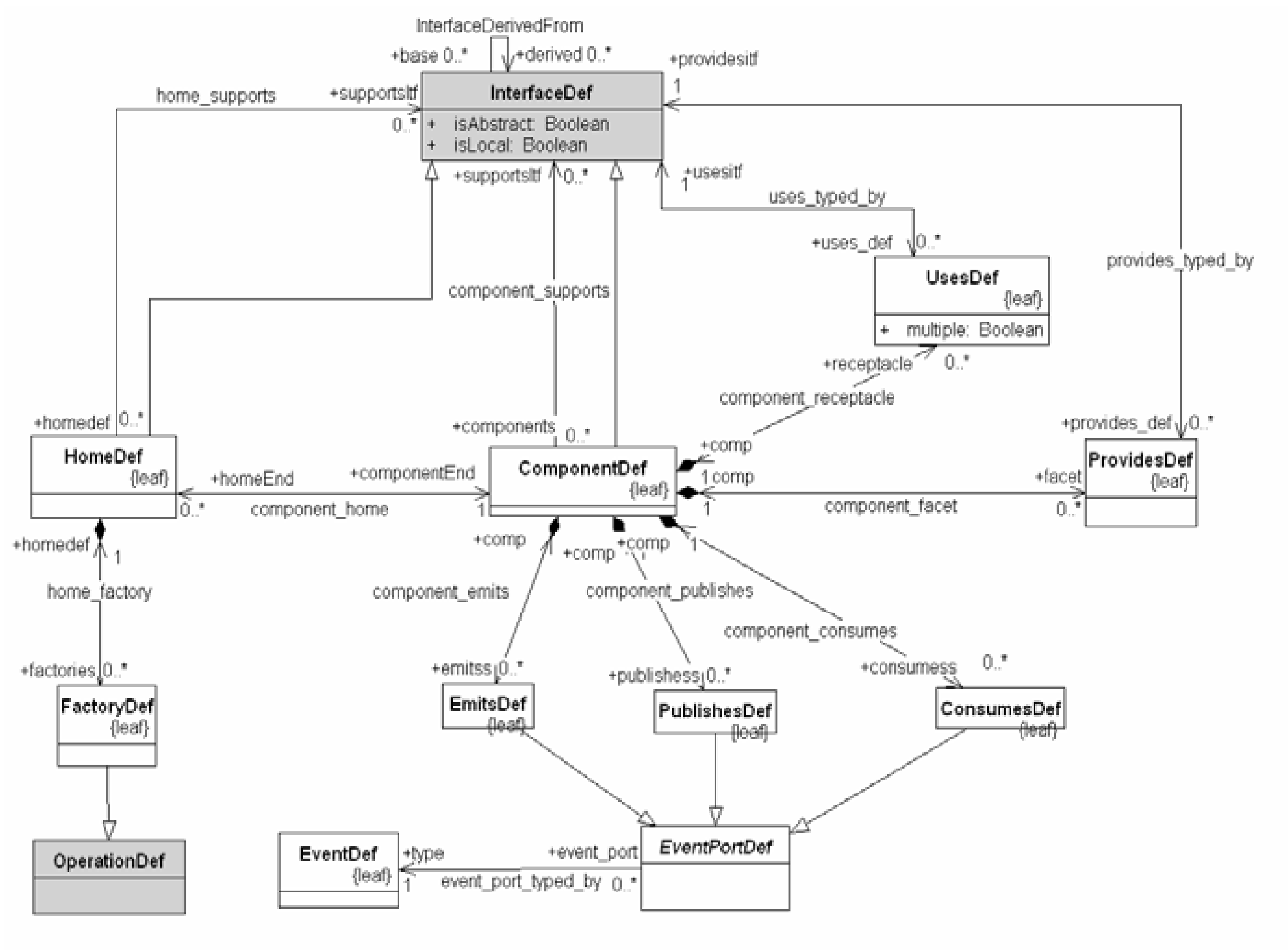

Figure 8.32 - ComponentIDL metamodel for the Lightweight CCM

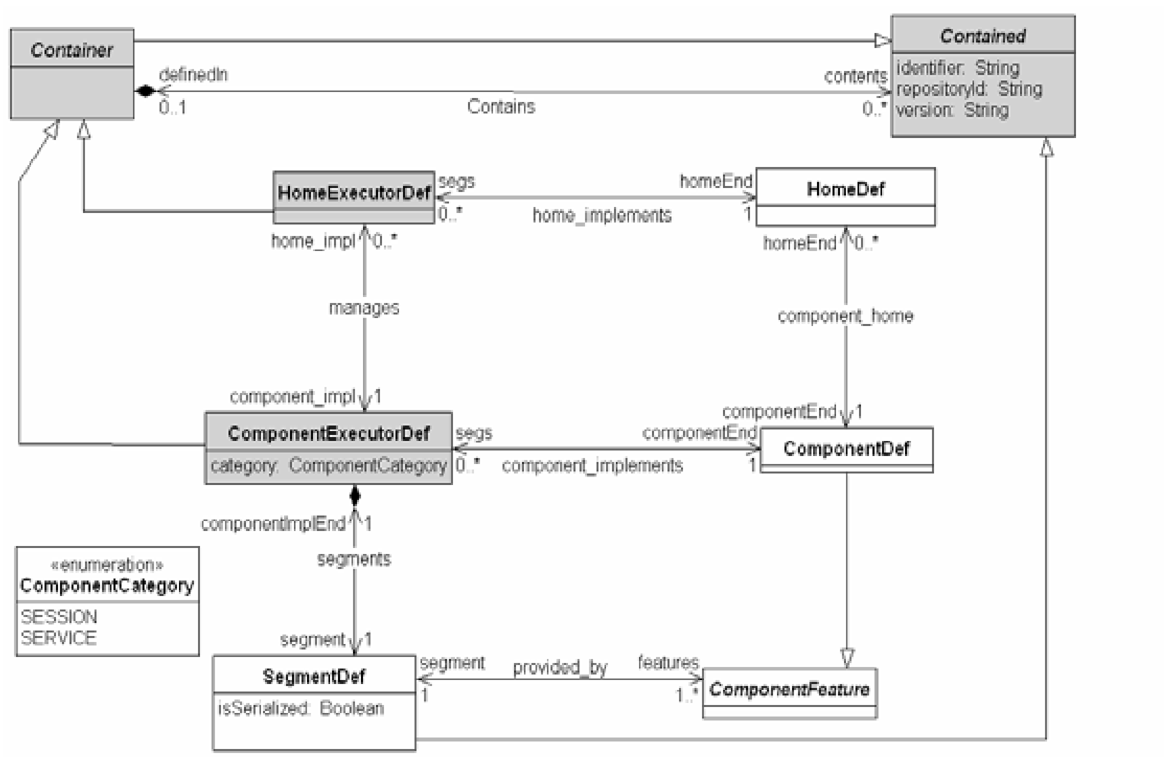

Figure 8.33 - CIF metamodel for the Lightweight CCM

8.7 Differences and Migrations between CORBA based Profiles

The UML Profile for CORBA (formal/02-04-01) specification (CORBA Profile) provides a standard means for expressing semantics of CORBA IDL using UML 1.3 notation and support for expressing these semantics with UML tools. The profile doesn't provide any means for expressing semantics of the CCM concepts like component or port. The profile doesn't define any MOF-based CORBA IDL metamodel.

The UML Profile for CORBA Component Model (CCM) specification (formal/05-07-06) provides a standard means for expressing both: pure CORBA and CCM-based applications, but using UML 1.5 notation. It is specified to work with MOF repositories since the profile defines a MOF-based CORBA IDL and CCM metamodel. This profile is based on the UML Profile for CORBA, extends it to the component-based semantics, and defines how to represent these semantics using UML 1.5. There are no migration rules from the UML Profile for CORBA to the UML Profile for CCM: all representations for CORBA IDL (including all data types, CORBA module and interface) were adopted for this profile.

This specification (CORBA&CCM Profile) provides a standard means for expressing both: pure CORBA and CCM-based applications using UML 2 notation. UML 2 facilitates and simplifies representation of many concepts needed to represent a pure CORBA or CORBA Components. The profile updates the MOF-based CCM metamodel and extends this metamodel to QoS and Deployment concepts. Due to various differences between UML 1.x and UML 2.x versions and new concepts in UML 2.x some migration rules were defined (see Table and description below). These rules provide an ability to automatically transform UML 1.x models based on the CORBA Profile to UML 2.x models based on the profile defined in this specification.

This specification is intended to replace the existing UML Profile for CORBA (formal/02-04-01) and UML Profile for CCM (formal/05-07-06) specifications.

Table 8.7 summarizes the main concepts of CORBA and CCM and gives an overview how mentioned above three specifications deals with these concepts. Additionally, the table shows all differences between profiles and identifies were some clarifications are needed for the successfully migration from the UML 1.x to UML2.x profile definition.

Table 8.7 - Differences between Profiles

Concepts	**CORBA Profile (UML 1.3)**	**CCM Profile (UML 1.5)**	**CORBA&CCM Profile (UML 2.1)**
Module	Package	Package	Package
Interface	Class	Class	Interface
Value	Class	Class	Interface
Constant	Attribute	Attribute	Property (Attribute)
Primitive Types	Data Type	Data Type	Data Type
Union	Class	Class	Data Type
Struct	Class	Class	Data Type
Exception	Exception	Exception	Data Type
Enum	Class	Class	Enumeration
Sequence	Class	Class	Data Type
Array	Class	Class	Data Type
AnonymousSequence	Class	Class	Data Type
AnonymousArray	Class	Class	Data Type
TypeDef	Class or Data Type	Class or Data Type	Data Type
Component		Class	Class
Facet (provided port)		Association between a component and its provided interface	Port
Receptacles (used port)		Association between a component and a used interface	Port
Event port (published, emitted or consumed)		Association between a component and its (published, emitted or consumed) event	Port
Event		Class	Interface
Stream			Interface

Concepts	CORBA Profile (UML 1.3)	CCM Profile (UML 1.5)	CORBA&CCM Profile (UML 2.1)
Stream port (source of sink)			Port
Home		Class	Interface
Component Executor		Class	Class
Home Executor		Class	Class
Composition			Component
Segment		Class	Part (Property)
Requirement			Class
File (contained, dependent, IDL or component)			Artifact
Assembly package			Package
Component package			Package
Monolithic implementation			Component
Collocation (host or process)			Collaboration
Configuration			Collaboration
QoS Binding			Comment

From the table above the following migrations from the CCM Profile (incl. CORBA Profile) to CORBA&CCM Profile have been identified:

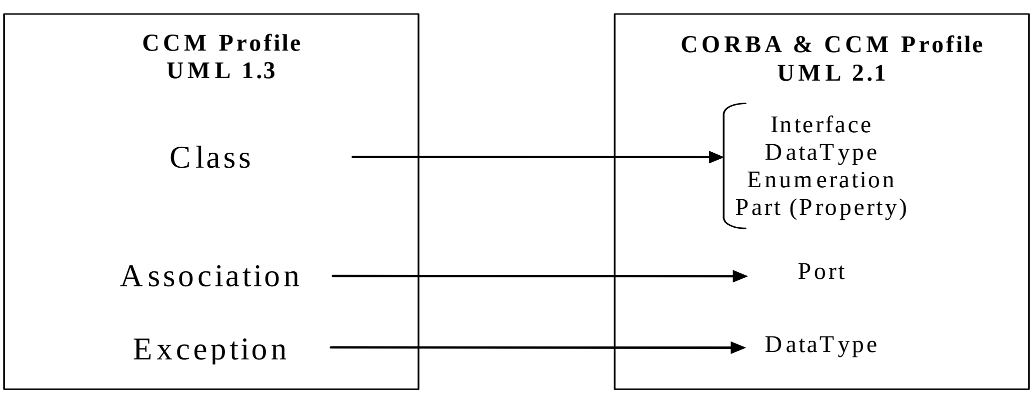

Figure 8.34 - Profile mappings

From the table and figure above following mappings (see also Figure 8.34) have been identified and described below:

- Class2Interface
- Class2DataType
- Class2Part(Property)
- Class2Enumeration
- Association2Port
- Exception2DataType

Class2Interface

The UML1.3 metamodel element "Interface" was inappropriate for modeling an IDL interface, as it may not have Attributes or Associations that can be navigated from the Interface. Therefore, the metaclass Class was taken to represent CORBA interface. The UML 2 metamodel element "Interface" provides all features for the representation of CORBA Interface. Mapping between these two metaclasses is simply described below:

Class2Interface (cl, itf)
FORALL UML1Class cl WHERE c.stereotype = "CORBAInterface" || "CORBAHome“
CREATE UML2Interface itf
SETTING itf.stereotype = cl.stereotype, itf.name = cl.name, itf.attribute = cl.attribute, itf.operation = cl.opertaion, itf.tag.isLocal = cl.tag.isLocal;

Class2DataType

The CORBA profile uses the metamodel element "DataType" only for the representation of CORBA Primitive types like short or string and TypeDefs, for user-defined Types like struct or sequence using of DataType was not possible, since DataTypes was not allowed to contain any Attributes, and Attributes are the best way to model struct/union members. The UML2 DataType may contain attributes to support the modeling of structured data types. Mapping between these two metaclasses is simply described below:

Class2DataType (cl, dt)
FORALL UML1Class cl WHERE cl.stereotype = ("CORBAStruct" || "CORBAUnion" || "CORBASequence" || "CORBAArray")
CREATE UML2DataType dt
SETTING dt.stereotype = cl.stereotype, dt.name = cl.name, dt.attribute = cl.attribute;

Class2Part

CCM segment is a set of artifacts, where each artifact is a physical part of the component executor and provides at least one facet. Each segment encapsulates independent state and is capable of being independently activated. The CCM Profile uses the metamodel element Class to model a segmented implementation structure for a component implementation (executor). UML 2 provides a new concept Part (metamodel element Property from InternalStructures). A part declares that an instance of the classifier may contain a set of instances by composition.

Class2Part (cl, part)
FORALL UML1Assoc ass LINKING UML1Class cl, UML1Class seg
WHERE cl.stereotype = "CORBAComponentImpl" AND seg.stereotype = "CORBASegment"
CREATE UML2Class cl2

SETTING cl2. stereotype = "CORBAComponentExecutor", cl2.name = cl.name
CREATE UML2Property part IN cl2
SETTING part.stereotype =seg.stereotype, part.name = seg.name, part.isSerialized= seg. isSerialized;

Class2Enumeration

The CORBA Profile uses the UML Class to represent a CORBA IDL enum type. Each element of the enum type is represented as an UML Attribute of the UML Class, with the same name as the enum element. UML 2 metamodel provides a metamodel element Enumeration for modeling such data types like IDL enum, whose instances may be any of a number of user-defined enumeration literals.

Class2Enumeration (cl, en)
FORALL UML1Class cl WHERE cl.stereotype = "CORBAEnum"
CREATE UML2Enumeration en
SETTING en.stereotype = cl.stereotype, en.name = cl.name, en.attribute = cl.attribute;

Association2Port

The metaclass Port is a new metamodel element, which has been added to UML 2. A port is a property of a UML classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. A Port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment. Due to missing port concept in the UML 1.x metamodel, CCM Profile uses metaclass Association for representing component ports in CCM. This has been changed in the current specification and the UML 2 port definition used for modeling of CCM component ports.

Association2Port(ass, port)
FORALL UML1Assoc ass LINKING UML1Class cl1, UML1Class cl2
WHERE cl1.stereotype = "CORBAComponent" AND cl2.stereotype = ("CORBAInterface" || "CORBAEvent")
CREATE UML2Port pt
SETTING pt. stereotype =ass.stereotype, pt.name = ass.name;

Exception2DataType

In CORBA Profile an IDL exception is represented by UML Exceptions (from CommonBehavior). In the UML 1.x metamodel, metaclass Exception is derived from metaclass Signal. UML 2 metamodel doesn't contain a metaclass Exception, exceptions that may be raised during an invocation of an operation are represented by an abstract metamodel element Type, which serves as a constraint on the range of values represented by a typed element. CORBA&CCM Profile represents an IDL exception by UML DataType element.

Exception2DataType(ex, dt)
FORALL UML1Exception ex WHERE ex.stereotype = "CORBAException"
CREATE UML2DataType dt
SETTING dt. stereotype =ex.stereotype, dt.name = ex.name, dt.attribute=ex.attribute;

9 Profile Illustration

9.1 Example Scenario Description

The “Simulation” example contains a set of components for simulating an Air Traffic Management (ATM) scenario in a very simplified way. The main purpose is to demonstrate the general usage of a graphical interface framework inside of the components while using a real world example context (simulation of ATM).

In the “Simulation” scenario there could be a number of planes, which are tracked by radar station whenever the planes are in their area of observation. Since the radar stations have only a limited area of observation and are located at different geographical positions it is important to combine the information that is provided by each of the radar stations into one single picture.

The example contains the following CORBA component types:

- **Plane**: This component represents a plane in the air that can be tracked by a radar station. It has a graphical user interface to receive commands regarding the speed and the heading of the plane. A plane component has a receptacle of type PlaneInput that is used to provide the current position of the plane to the simulation server. This demonstrates the usages of a synchronous communication.
- **SimulationServer**: This component should be instantiated once in a simulation scenario. It retrieves the position of all planes in a synchronous manner. Radar station can get information about the planes that are in their area of observation. The simulation server computes this based on the location position provided by the radar stations. This component does not expose a graphical user interface.
- **Radar**: This component simulates a radar station. The component acquires the information about the planes that are currently in its area of observation by sending a synchronous request to the simulation server. In this request the radar station provides its own location. The information about the planes is then presented to the user in a graphical form. Furthermore, the radar station provides the information about the planes in the area of observation to the TAPDisplay component in an asynchronous manner.
- **TAPDisplay**: This component obtains information from all radar station about the position of the radar station and the planes in the area of observation. The TAPDisplay presents the information gathered from all radar stations to the user in a single view.

9.2 Type Definition

The example uses the following IDL3 basic types and exceptions:

9.2.1 IDL Notation

```
module Simulation
{

        /* Position e.g. of an airplane */
    struct Position{
        double longitude;
        double latitude;
        double altitude;};

        /* Position of a radar Contact */
```

```
struct PolarPosition{
   double angle;
   double distance;};

struct RadarObject{
   string identifier;
   Position position;};
```

/* Transponder information */

```
struct TransponderObject{
   string identifier;
   double altitude;};
```

/* a sequence of radar contacts */

```
typedef sequence<RadarObject> RadarData;
```

/* List of radar contacts submitted to base stations */

```
eventtype RadarEvent {
   public string radar_identifier;
   public Position radar_position;
   public RadarData radardata;
   public double radius;};
```

/* dynamic information about airplane position */

/* possibly from FlightGear */

```
interface PlaneInput {
   void set_position(in string identifier, in Position current_position);};
```

/* plane */

/* only needed if FlightGear is not available */

```
component Plane {
   attribute string identifier;
   attribute double initial_longitude;
   attribute double initial_latitude;
   attribute double initial_altitude;
   attribute double initial_course;
   attribute double speed;
   uses PlaneInput sim_server;};
home PlaneHome manages Plane {};

interface RetrieveRadarData {
```

/* callculates the List of radar contacts visible for a given position of a Radar */

```
   RadarData get_data(in Position radar_position, in double radius);};

component SimulationServer{
   provides PlaneInput the_input;
   provides RetrieveRadarData radar_output;};
home SimulationServerHome manages SimulationServer {};

component Radar {
   attribute string radar_identifier;
   attribute double longitude;
```

```
      attribute double latitude;
      attribute double radius;
      attribute double pixel_radius;
      uses RetrieveRadarData sim_server;
     publishes RadarEvent to_tac_display;};
   home RadarHome manages Radar {};

   component TAPDisplay {
      attribute string identifier;
      attribute double longitude;
      attribute double latitude;
      attribute double horizontal_range;
      attribute double vertical_range
      attribute double horizontal_pixels;
      attribute double vertical_pixels;
      consumes RadarEvent from_radar;};
   home TAPDisplayHome manages TAPDisplay {};
};
```

9.2.2 CIDL Notation

```
module Simulation
{
  composition session PlaneImpl {
    home executor PlaneHomeImpl {
      implements PlaneHome;
      manages PlaneSessionImpl;};};

  composition session SimulationServerImpl {
    home executor SimulationServerHomeImpl {
      implements SimulationServerHome;

    manages SimulationServerSessionImpl;};};

  composition session RadarImpl {
    home executor RadarHomeImpl {
      implements RadarHome;
      manages RadarSessionImpl;};};

  composition session TAPDisplayImpl {
    home executor TAPDisplayHomeImpl {
      implements TAPDisplayHome;
      manages TAPDisplaySessionImpl;};};
};
```

9.3 UML Example Diagrams

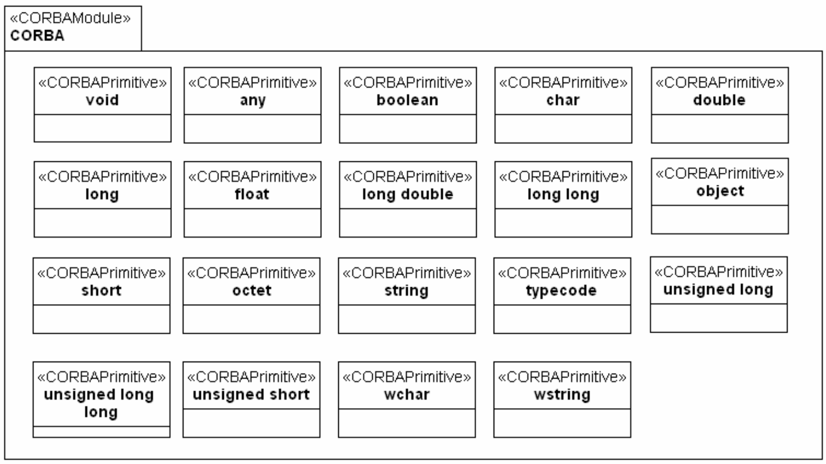

Figure 9.1 - CORBA Package

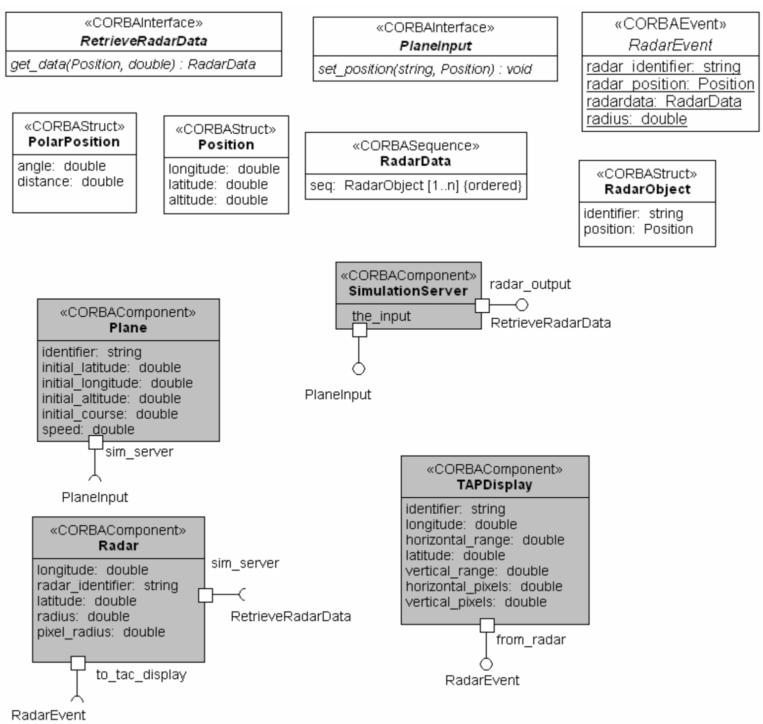

Figure 9.2 - User-defined data types, interfaces and components

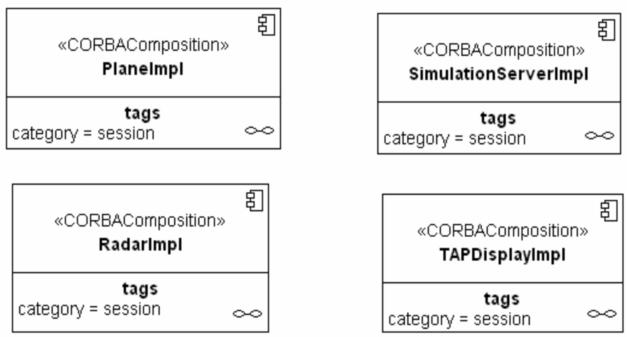

Figure 9.3 - Compositions

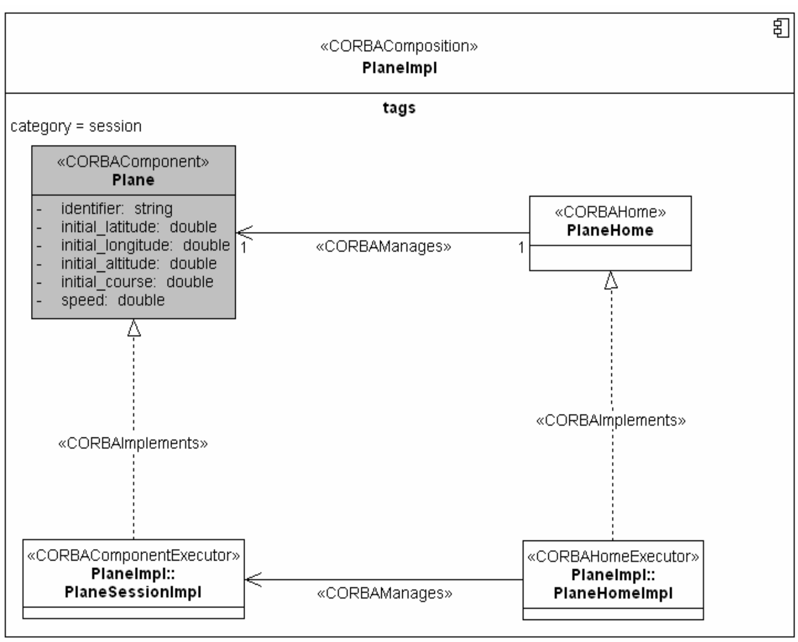

Figure 9.4 - Composition description for Plane component

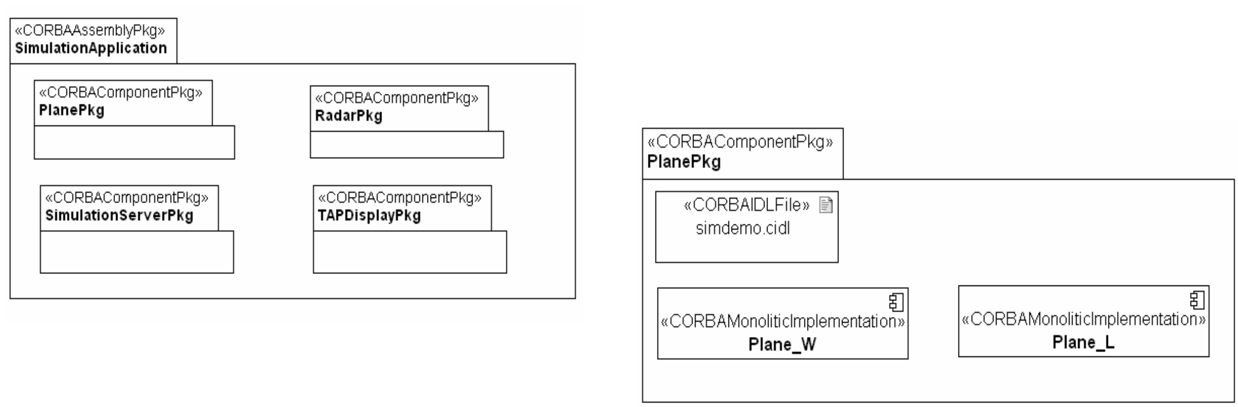

Figure 9.5 - Assembly package and component package content for the Plane component

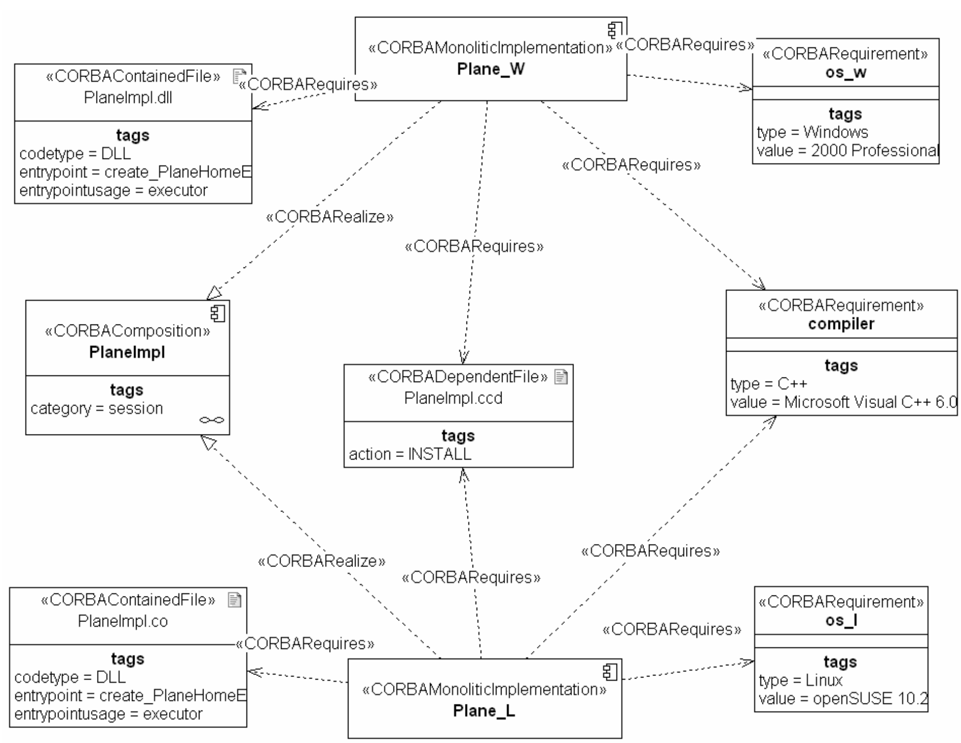

Figure 9.6 - Component Implementation description

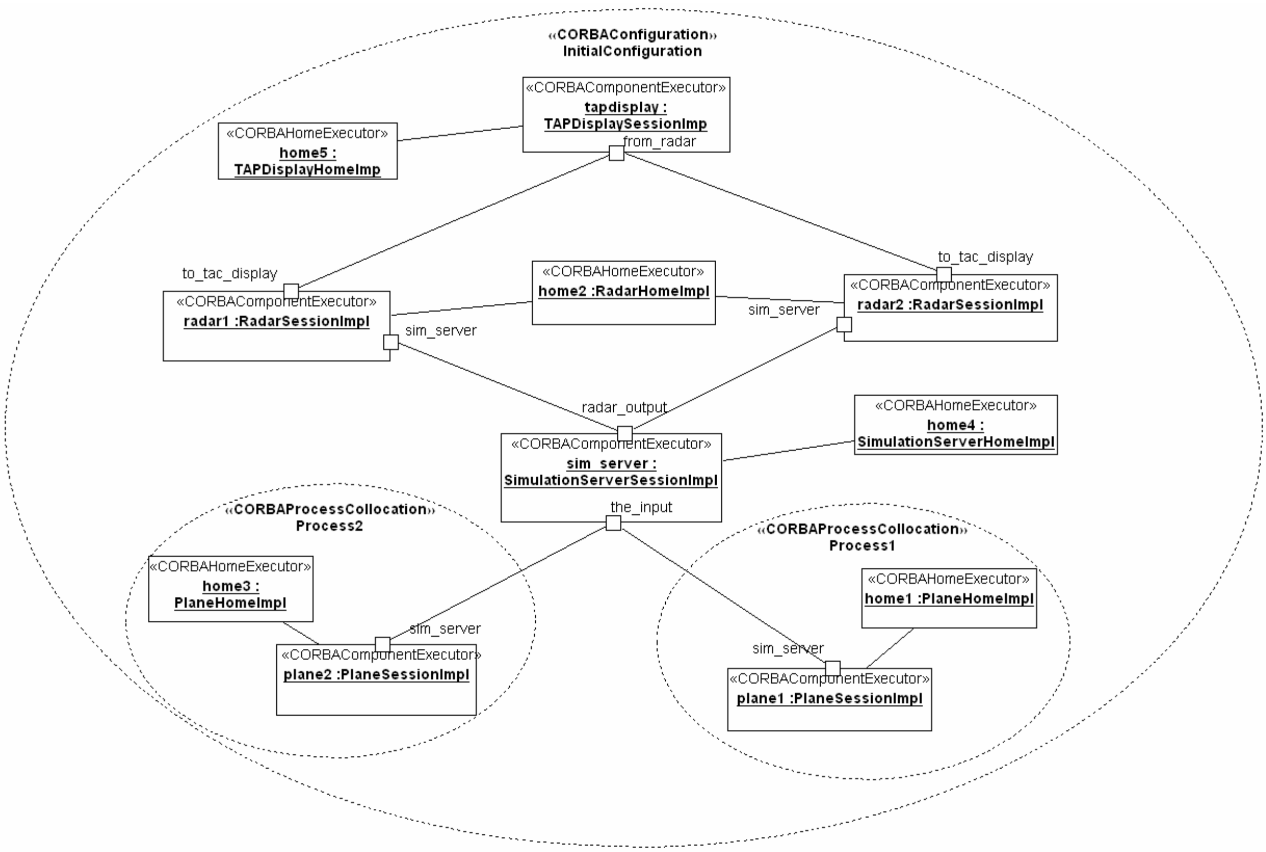

Figure 9.7 - Initial Configuration Description

Annex A   References

[1] Meta Object Facility (MOF) Core Specification, Version 2.0, OMG document formal/2006-01-01

[2] CORBA Components Specification, OMG document formal/02-06-65

[3] Unified Modeling Language (UML) Specification, Version 2.0, OMG document formal/03-03-01

[4] Unified Modeling Language (UML) Superstructure Specification, Version 2.1.1: OMG document formal/07-02-05

[5] Unified Modeling Language (UML) Infrastructure Specification, Version 2.1.1: OMG document formal/07-02-06:

[6] UML 2.1.1 XMI file: OMG document ptc/06-10-06

[7] UML Diagram Interchange Specification: OMG document formal/06-04-04

[8] UML OCL Specification, Version 2.0: OMG document formal/06-05-01

[9] CORBA Component Model Specification, Version 4.0: OMG document formal/2006-04-01

[10] The UML Profile for CORBA, OMG document formal/02-04-01

INDEX