INTERNATIONAL TELECOMMUNICATION UNION

ITU-T Z.130

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(07/2003)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Extended Object Definition Language (eODL)

Extended Object Definition Language (eODL): Techniques for distributed software component development – Conceptual foundation, notations and technology mappings

ITU-T Recommendation Z.130

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
Extended Object Definition Language (eODL)	**Z.130–Z.139**
Testing and Test Control Notation (TTCN)	Z.140–Z.149
User Requirements Notation (URN)	Z.150–Z.159
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-computer interfaces for the management of telecommunications networks	Z.360–Z.369
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Distributed processing environment	Z.600–Z.609

For further details, please refer to the list of ITU-T Recommendations.

ITU-T Recommendation Z.130

Extended Object Definition Language (eODL): Techniques for distributed software component development – Conceptual foundation, notations and technology mappings

Summary

This Recommendation is intended for designers, implementers and managers of distributed systems, and tool developers that provide tools to support distributed systems.

This Recommendation specifies the ITU Extended Object Definition Language (ITU-eODL). ITU-eODL is used for a component-oriented development of distributed systems from the perspectives of four different but related views: the computational, implementation, deployment, and target environment view. Each view is connected with a specific modelling goal expressed by dedicated abstraction concepts. Computational object types with (operational, stream, signal) interfaces and ports are the main computational view concepts which describe distributed software components abstractly in terms of their potential interfaces. Artefacts as abstractions of concrete programming language contexts and their relations to interfaces form the implementation view. The deployment view describes software entities (software components) in binary representation and the computational entities realized by them. The target environment view provides modelling concepts of a physical network onto which the deployment of the software components shall be made. All concepts of the views are related to each other. These relations form an essential base for techniques and tools that support the software development process from design via implementation and integration to deployment. The test phase is not yet captured by this Recommendation.

ITU-eODL is an extension of the ITU Object Definition Language ITU-ODL [1] and supersedes it. Originally ITU-ODL was designed as an extension of ODP-IDL [9] and defined computational concepts based on ODP [2], [3] terminology. eODL follows this principle. However, definitions are based on a metamodel rather than the traditional abstract syntax approach. One advantage of the metamodel approach is to allow use of MOF [4] related tools to support the automation of model transitions between the different software development phases. Another benefit is the ability to instantiate concrete models from the metamodel, which can be represented by existing languages, so an integration of different design approaches can be achieved.

The readers of this Recommendation are expected to be familiar with IDL [5], UML [11], MOF.

The definition of eODL is supported by the following annexes and appendices:

- Annex A introduces a textual syntax for eODL, intended to be used for the representation of eODL specifications. The syntax is defined using the EBNF style.
- Annex B defines the mapping between the eODL metamodel and the textual syntax defined in Annex A.
- Annex C provides a mapping from eODL to ITU SDL-2000 that allows an eODL model to be automatically transformed to a SDL-2000 model.
- Annex D contains a software reference to the XML representation [12] of the eODL metamodel according to the XML meta interchange format (XMI) [6]. It is provided in a separate file in order to allow import and processing of the eODL metamodel by UML tools.
- Clause 1 provides an overview of how eODL is used by designers, implementers and managers of a distributed system. A concrete example of the use is given in Appendix I.
- Appendix II describes the overall development process when using eODL and possible tool support.

Source

ITU-T Recommendation Z.130 was approved on 22 July 2003 by ITU-T Study Group 17 (2001-2004) under the ITU-T Recommendation A.8 procedure.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure e.g. interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

ITU-T Recommendation Z.130

Extended Object Definition Language (eODL): Techniques for distributed software component development – Conceptual foundation, notations and technology mappings

1 Scope

The provision of efficient techniques and of tool support for the development and engineering of distributed systems is a key enabling factor for the further evolution of Information Technology. Telecommunication systems are special distributed systems consisting of components which are distributed across networks and have to cope with concurrency, autonomy, synchronization, and communication aspects. The development of highly efficient and scalable systems is a complex and complicated task, where tools have to support all phases of the development process – from requirements capturing over design and implementation to integration, test and deployment.

Code generation out of object-oriented design models leads to reusable, executable components. Such components integrate runtime environment and middleware platform technology dependent aspects with the enterprise specific object-oriented design model. Each software component has a physical representation (e.g., binary file), which has to be available for execution on a special node of a distributed system. The main focus of this Recommendation is the design of such components.

Techniques for the development of distributed systems contribute significantly to a reduction of the time to market of distributed applications and telecommunication services. An appropriate treatment of all kinds of communication aspects lies in the very nature of the targeted application domain. These aspects span from transactional requirements on object interactions over quality of service issues to security policies. Taking into account the broad acceptance of object middleware technology, middleware platforms provide an ideal implementation environment for such designs. Of these are plain CORBA [5], CORBA Components [7] and other distributed processing platforms.

This Recommendation is targeted to all software development processes addressing the following phases of the software life cycle:

- design phase;
- implementation phase;
- integration phase;
- operational phase.

This Recommendation does not address the requirement capturing phase.

The special emphasis of this Recommendation lies on the technological support of the transitions between phases to achieve their automation. The key technology is a model driven approach which is based on a well-defined metamodel (see [11]). This metamodel allows integration of several existing design languages like SDL [8], UML and CORBA-IDL. The metamodel is the definition of the concepts for the addressed phases of the software lifecycle. The models being instantiated on the basis of the metamodel can be represented using the existing languages. Since some concepts are not covered by an existing language, this Recommendation also defines a concrete syntax: eODL (Extended ODL). The metamodel-based approach replaces the well-known abstract syntax approach for the definition of languages. ITU-eODL is a revision of ITU-ODL. The syntax definition is given by Annex A.

Consequently, the metamodel is independent of a specific design notation. Design models can be developed applying different notations, but are based on the same concepts. Design information can be exchanged on the basis of the metamodel. Both the notation and the metamodel are independent

of a specific runtime environment. The same design can be mapped onto different target environments. This enables high flexibility and is also important for the aspect of reuse of component design models.

The integration of this Recommendation in software development processes is depicted in Figure 1. Starting with a precise requirements definition, a design model is specified. This Recommendation defines concepts which enable different views of the design model. Each view covers different aspects of the system to be developed. The concepts of the metamodel allow development of models powerful enough to derive software component skeletons automatically. The skeletons form the starting point of the implementation phase, where they have to be completed by the business logic. In the integration phase the completed software components have to be integrated in the target environment.

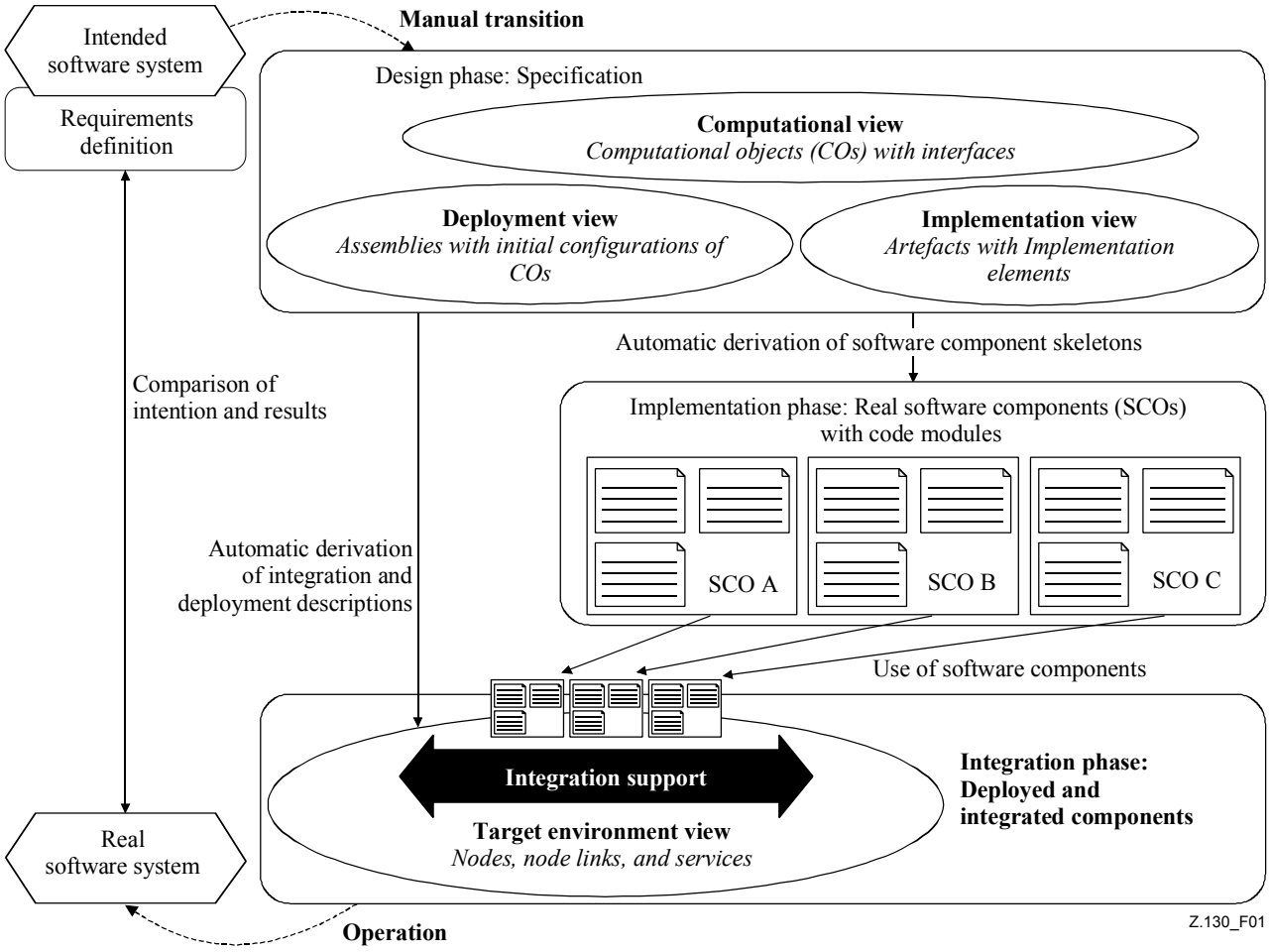


Figure 1/Z.130 – Software development

This Recommendation also contains concepts that allow the description of the topology and properties of the target environment. Together with the automatically generated deployment and integration descriptions stemming from the design phase, the target environment description enables the automation of the deployment. After the integration phase, the developed software system can be put into operation.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[1] ITU-T Recommendation Z.130 (1999), *ITU object definition language*.

[2] ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, *Information technology – Open Distributed Processing – Reference Model: Foundations*.

[3] ITU-T Recommendation X.903 (1995) | ISO/IEC 10746-3:1996, *Information technology – Open distributed processing – Reference Model: Architecture*.

[4] OMG Document formal/00-04-03, *Meta Object Facility (MOF) Specification*, Version 1.3.

[5] OMG Document formal/01-02-33, *The Common Object Request Broker Architecture and Specification*, Revision 2.4.2.

[6] OMG Document formal/00-11-02, *XML Metadata Interchange (XMI) Specification*, Version 1.1.

[7] OMG Document formal/02-06-65, *CORBA Component*, Version 3.0.

[8] ITU-T Recommendation Z.100 (2002), *Specification and Description Language (SDL)*.

[9] ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1999, *Information technology – Open Distributed Processing – Interface Definition Language*.

[10] ITU-T Recommendation Z.600 (2000), *Distributed processing environment architecture*.

3 Abbreviations

This Recommendation uses the following abbreviations:

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CCM	CORBA Component Model
CO	Computational Object
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CWM	Common Warehouse Metamodel
DTD	Document Type Definition for XML
EBNF	Extended Backus-Naur Form
EJB	Enterprise JavaBeans™
IDL	Interface Definition Language
MDA	Model Driven Architecture
MOF	Meta Object Facility
OCL	Object Constraint Language

ODL	Object Definition Language
OMG	Object Management Group
OSD	Open Software Description
RM-ODP	Reference Model for Open Distributed Processing
SDL	Specification and Description Language
TINA	Telecommunication Information Networking Architecture
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

4 Definitions

This Recommendation defines the following terms:

4.1 **artefact**: Abstraction of concrete programming language constructs, e.g., a class in case of object-oriented languages (realized in the form of code modules), enclosed by a **software component**. An **artefact** instance **realizes** (in a model sense) the state, behaviour and identity of a **CO**.

4.2 **assembly**: Description of a distributed software system by the set of all participating **CO types** and the **initial configuration**. (Used by CCM.)

4.3 **exception**: Special kind of operation **termination** in case of errors. (Defined by RM-ODP.)

4.4 **class**: Concept for the classification of objects. On the basis of a class description, objects can be instantiated. (Defined by MOF.)

4.5 **connection**: Concept for the exchange of **interface references**, which belong to *port* definitions according to their special definition kind (*use* or *provide* relation). (Defined as computational binding by RM-ODP; used by CCM.)

4.6 **component architecture**: **Distributed processing environment** which supports the interaction of distributed **software components**.

4.7 **target platform**: **Component architecture** with support of **deployment** and distributed execution, where components are intended to be deployed.

4.8 **computational object (CO)**: Functional unit that results from a functional decomposition of the software system being modelled. (Defined by RM-ODP.)

4.9 **computational object type (CO type)**: Template for the instantiation of **computational objects**. (Defined by RM-ODP; identified by CCM as CORBA component.)

4.10 **consume**: Concept for modelling potential signal reception. (Used by CCM.)

4.11 **continuous medium**, **signal** and **operational interaction**: **Interaction** between **COs** using operational-, signal- or *continuous medium* **interaction elements**: **operation**, **attribute**, **consume**, **produce**, **sink**, **source**. (Defined by RM-ODP and TINA (operational and *continuous media* interaction).)

4.12 **data type**: A prescription of permissible structure, contents and behaviour of data; it is an element of a data type model (i.e., CORBA-IDL) which is a base of information models. (Defined by CORBA.)

4.13 **distributed processing environment**: Technological base, supporting interactions between objects of a distributed system. (Defined by TINA.)

4.14 deployment: The process of making physical representations of **software components** available on **nodes**, installing them so that they are ready for execution and for setting up the **initial configuration**.

4.15 implementation element: Relation between an **interaction element** and an **artefact**, where an **artefact** instance is responsible for behaviour of the **interaction element**.

4.16 implements: Relation between **artefact**s and **CO types**, where **artefact** instances **realize** the behaviour of the **CO type**.

4.17 initial CO: **CO** which is created at the beginning of the **runtime** of a distributed software system.

4.18 initial configuration: Set of **initial COs** and **initial connections**. (Used by CCM.)

4.19 initial connection: A binding which is established initially at the beginning of the **runtime** of a distributed software system. (Used by CCM.)

4.20 instantiation policy: Policy-based description of instantiation of various implementation concepts modelled by **artefacts**.

4.21 interaction: Action in which the environment of an object is involved. (Defined by RM-ODP.)

4.22 interaction element: Generalization of the concepts **operation**, **attribute**, **sink**, **source**, **consume** and **produce**.

4.23 interface: Referencible aggregation of possible interactions of a **CO**. (Defined by RM-ODP.)

4.24 interface attribute: Special kind of operations as a shorthand for *get* and *set* operations for a given **data type**. (Used by CORBA.)

4.25 interface reference: Reference to an **interface**. (Corresponds to CORBA object reference.)

4.26 interface type: Description of a set of **interaction elements** as named and identifiable endpoints of possible communication. (Defined by RM-ODP; corresponds to OMG IDL interface.)

4.27 media set: Aggregation of media.

4.28 medium type: Declaration to be used for coding, transmission and decoding of the data on a **medium**.

4.29 medium: Atomic unidirectional continuous stream of data. (Replaces the stream notation from ITU-T Rec. Z.600 [10].)

4.30 metamodel: Definition of modelling concepts for the construction of models of a specific domain. (Defined by MOF.)

4.31 meta-metamodel: Definition of modelling concepts for the construction of **metamodels**. (Defined by MOF.)

4.32 multiple port: **Port** which dynamically supports registration and retrieval of multiple **interface references**. (Defined by CCM.)

4.33 name space: Concept to structure identifiers of model elements. (Defined by RM-ODP.)

4.34 node: Device used for interpretation of the code modules of a **software component**. (Defined by RM-ODP.)

4.35 object: Model of an entity, where an entity is any phenomenon of interest in the examined domain; an object has an identity, state and behaviour. (Defined by RM-ODP.)

4.36 operation: Element of an operational interaction, described by a set of parameters and possible terminations. (Defined by RM-ODP, CORBA.)

4.37 parameter: Element of the invocation of an operation, described by the direction of an information exchange and a **data type**. (Defined by CORBA.)

4.38 port: Entity for registration and retrieval of **interface references** of a **CO**. (Defined by CCM.)

4.39 produce: **Interaction element** to send **signals**. (Defined by CCM.)

4.40 provided port: **Port** where **interface references** of the corresponding **CO** can be retrieved. (Defined by CCM.)

4.41 realize: Correspondence of **software components** to **CO types**. (Defined by UML.)

4.42 requires: Relation of an **interface type** and a **CO type** where the **COs** of this **CO type** require **interface references** of the **interface type** from the **CO** environment. (Defined by TINA.)

4.43 runtime: The time when a **software component** is executed.

4.44 signal: **Interaction element** for the asynchronous exchange of atomic messages. (Defined by RM-ODP.)

4.45 single port: **Port** where only a single **interface reference** can be registered and retrieved. (Defined by CCM.)

4.46 sink: **Interaction element** to receive a **media set**. (Defined by TINA.)

4.47 software component: An entity which consists of sequences of instructions (code modules), which is physically represented (in form of special data format), and which can be assembled to structured **software components** or to a software system; to enable the composition of **software components**, their functionality is provided via well-defined **interface types**.

During an execution of a **software component**, objects are incarnated as instances of classes (realized as code modules). (Defined by [16].)

4.48 software package: Package of **software components**. (Defined by CCM.)

4.49 source: **Interaction element** for sending a **media set**. (Defined by TINA.)

4.50 supports: Relation of an **interface type** and a **CO type**, where the **COs** of this **CO type** support **interface references** of the **interface type** from the **CO** environment. (Defined by TINA.)

4.51 termination: End of an invoked **operation**. (Defined by RM-ODP.)

4.52 signal parameter: Element of a **signal** to carry information; refers to a **data type**. (Defined by SDL.)

4.53 used port: **Port** where **interface references** can be registered. (Defined by CCM.)

5 Metamodel

A **metamodel** defines modelling concepts for the construction of models of a specific domain. The **metamodel** in this Recommendation is a Meta-Object Facility (MOF) compliant **metamodel**. The **metamodel** is described by means of UML class diagrams. The semantics is given in natural language. When needed, well-formedness rules are added. The MOF is the adopted standard for metamodelling in the OMG (Object Management Group). The MOF defines a generic framework for describing and representing meta-information.

The MOF defines a four-level architecture, depicted in Figure 2:

- In the M3 level, we find a single **meta-metamodel** (the MOF model) that defines the basic concepts needed to describe any **metamodel** in an object-oriented way. The basic constructs are: class, association, data type, class attribute and class inheritance.

- In the M2 level, we find **metamodels** (languages). A **metamodel** provides the abstractions that are needed to build models. It is described in terms of the M3 basic constructs (in practice the abstract syntax of a **metamodel** is provided as a collection of class diagrams). Examples of **metamodels** are UML, CWM (data warehouse), and the CCM **metamodel**.
- In the M1 level, we find *models*. These are described in terms of one of the **metamodels** defined at the M2 level. An example of a model is a network level model in UML that defines what a trail is.
- In the M0 level, we find data, which are instances of a model at the M1 level. A list of records representing trails is example data.

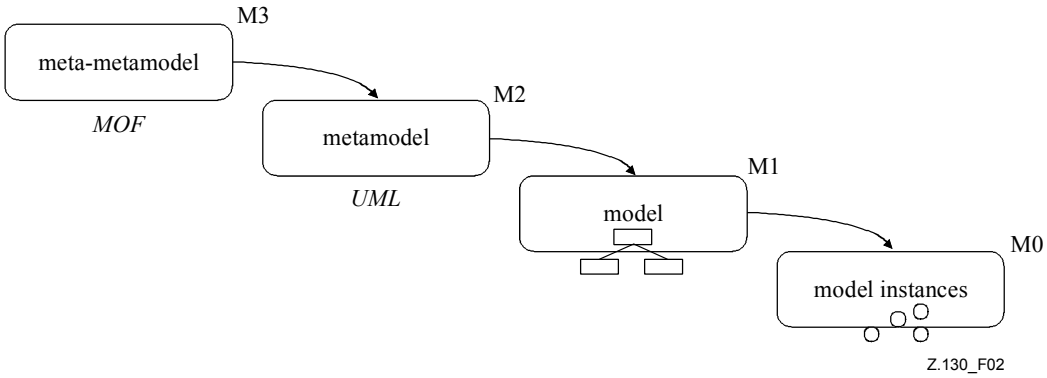


Figure 2/Z.130 – MOF levels

In addition to basic language constructs for the object-oriented description of **metamodels**, the MOF standardizes the **interface types** (OMG IDL interfaces) that can be used to operate on model entities. Furthermore, the related XMI [6] standard standardizes the way a model can be externalized in an XML [12] stream format. The XML vocabulary used for externalization depends only on the **metamodel** entities.

5.1 Definitions and conventions

This clause defines concepts of the MOF model (**meta-metamodel**) which are used to define the eODL concepts (**metamodel**). The notation used for the visibility of attributes and operations is introduced in Figure 3. It is consistently used in all UML figures. For further information, please refer to [4].

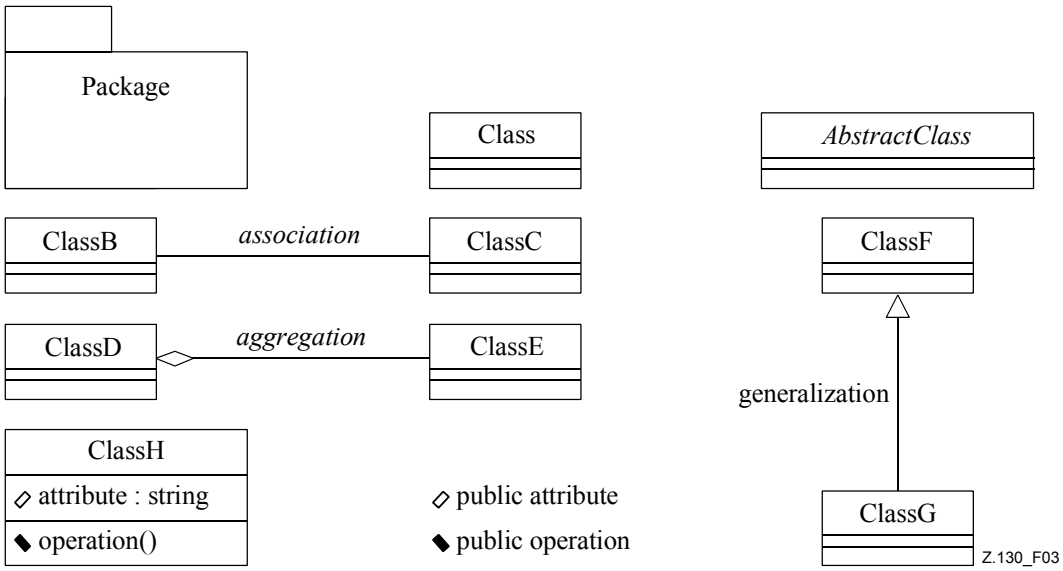


Figure 3/Z.130 – UML notation for MOF concepts

5.1.1 ***class and object***: A class is the description of a set of objects all having the same distinct class characteristics. Class is used for classification and serves as the basic concept for construction. An object is an instance of a certain class.

5.1.2 ***generalization***: A generalization is a unidirectional relation between two classes. Generalization associates the special and the general class. The special class inherits all characteristics of the general class.

5.1.3 ***association***: An association is a relation between two classes. In case there are instances of the two classes, an association may or may not be instantiated between them. The association is navigable from one involved object to the other.

5.1.4 ***aggregation***: An aggregation is a directed relationship between classes where the instances of the aggregated classes (parts) are considered to be contained in instances of the container class (i.e., the aggregating class). Its semantics is that of a 'has-a' relation. There is a distinction between strong and weak aggregation in regard to the life cycle of the parts and their container. In this Recommendation, always strong aggregation is applied. It is used for the reflection of composition respective decomposition.

The **metamodel** of eODL is defined using the UML notation for MOF (see Figure 4). The constraints and well-formedness rules which are part of this **metamodel** and essential for its semantics are provided as English text. The UML diagrams which appear as figures in the following clauses show only parts of the complete eODL **metamodel**. The explaining text and especially the constraints contained in the text have to be read in order to understand the semantics. Constraints which are already described as part of the **metamodel** of IDL are not mentioned again in this Recommendation. Please refer to [7] instead.

The complete **metamodel**, including all constraints, is referenced as an XMI stream in Annex D.

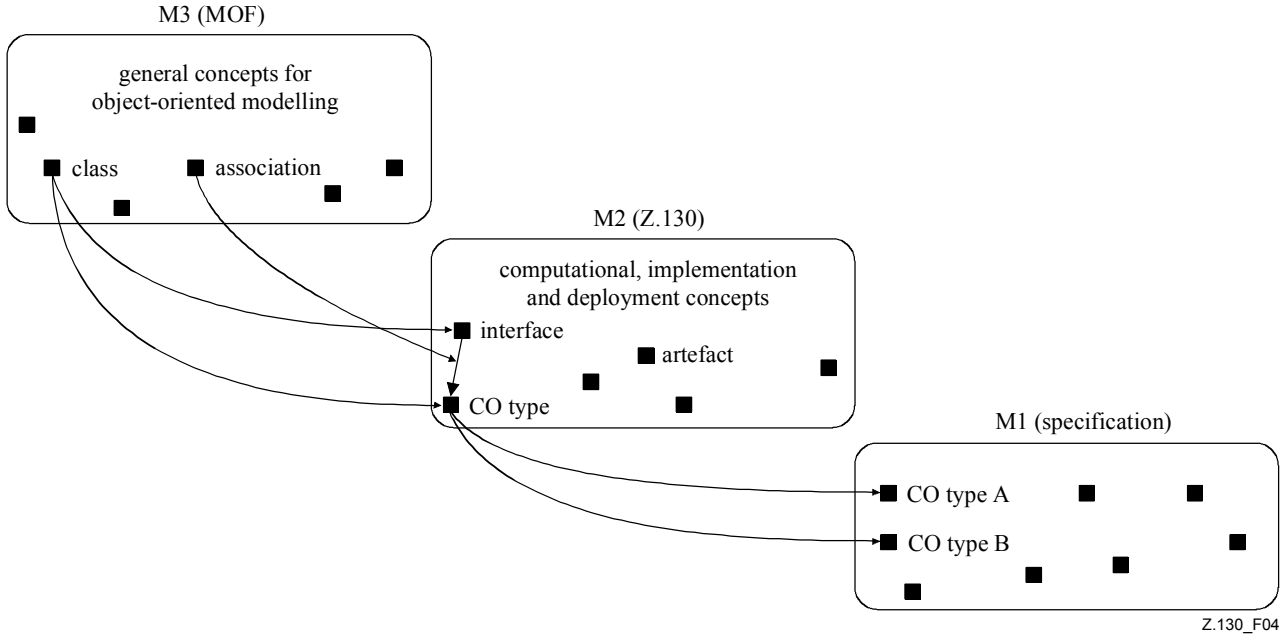


Figure 4/Z.130 – Object-oriented concepts

5.2 Naming and scoping

Naming and scoping rules are defined to enable the unambiguous identification of model elements.

The entire model forms a name scope. Each entity that forms a new scope is an instance of the abstract metaclass *Container*. Contained elements of a scope are instances of the abstract metaclass *Contained*. Defining the metaclasses *Container* and *Contained* as being abstract implies that all

instances are instances of derived non-abstract metaclasses. The *Container-Contained* relation is frequently used in the **metamodel**.

Each named entity (instance of metaclass *Contained*) has an identifier to denote the name. The identifier is an attribute of the metaclass *Contained*. The identifiers of two different named entities, which belong to the same *Container* (definedIn points to the same model element) must be different.

To allow pure scopes in a model, the metaclasses *ModuleDef* is introduced. *ModuleDef* is a part of the **metamodel** of CORBA-IDL on which the **metamodel** in this Recommendation is based. It is a concrete metaclass and can be instantiated. It has no further properties.

Each instance of *Container* forms a namespace. The generalization from *Contained* to *Container* expresses the ability of nesting name scopes (see Figure 5).

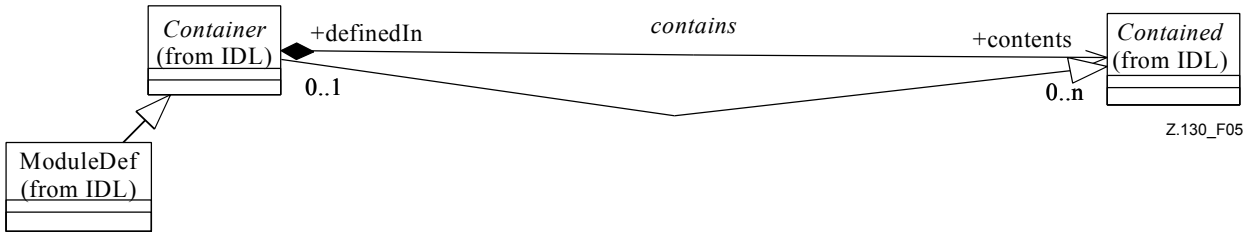


Figure 5/Z.130 – Naming and scoping

5.3 Computational concepts

5.3.1 Used CORBA-IDL concepts

In order to introduce **data types**, **operations**, **attributes**, **exceptions** and **interface types** as modelling concepts, the eODL **metamodel** is based on the **metamodel** of CORBA-IDL.

All these modelling concepts allow the definition of basic building blocks for computational specifications. One purpose of a computational specification is to define the signatures of **computational objects** (**CO**) at their **ports**. Since this Recommendation introduces a type-based modelling, **data types** are essential to describe such signatures.

5.3.1.1 Data types, Interface types, Operations, Attributes, Exceptions

Data types in models are instances of metaclasses which are derived from the abstract metaclass *IDLType*. This implies the inclusion of the whole CORBA-IDL data type system. Through the usage of the abstract metaclass *IDLType*, it is ensured that the data type system can be exchanged to ensure the openness of this Recommendation. Figure 6 shows a subset of the CORBA-IDL **metamodel** for **data types**. The **metamodel** together with its description can be found in [7].

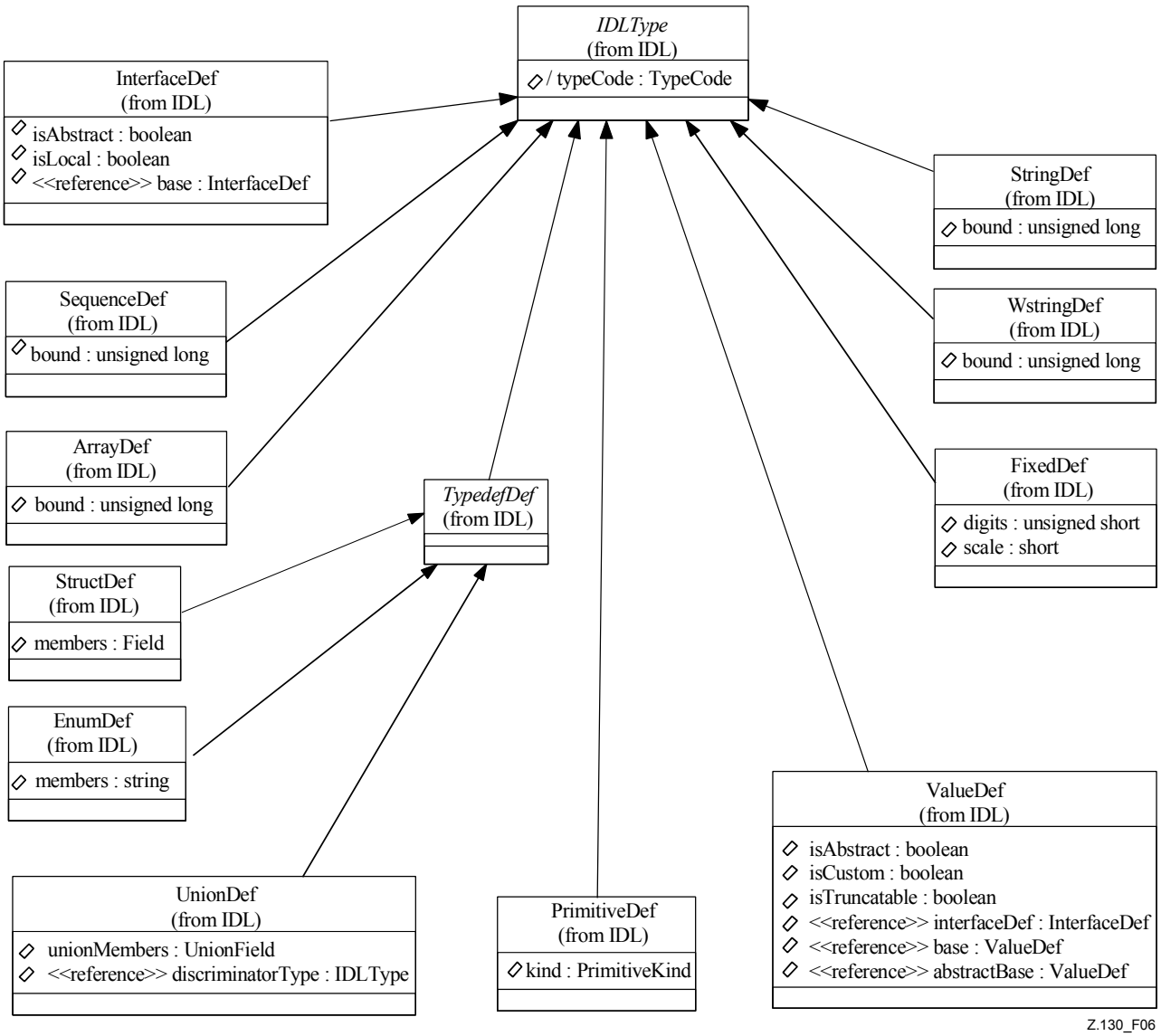


Figure 6/Z.130 – Data types

The CORBA data type system contains all commonly used primitive **data types** such as *long*, *float*, *char*, *string*, etc. Furthermore, there are concepts to describe structured **data types** like *array*, *struct*, *sequence*, *union*, etc. The **data type** *value* is an additional type which is used to pass objects using a object by value semantic. Like classes in programming languages, an instance of type *value* can aggregate **attributes** and **operations**.

In order to use other data type definitions different from those included in CORBA, the element ExternType is introduced. It is a specialization of the concept TypedefDef of CORBA. The attribute identifier refers here to an externally provided definition of the data type. See Figure 7.

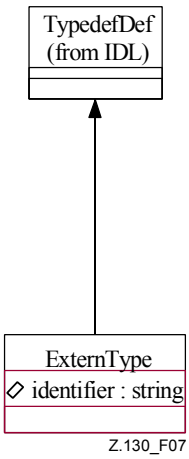


Figure 7/Z.130 – Extern type

Attributes, **operations**, **exceptions** and **parameters** are concepts required for the definition of operational interactions. An **operation** defined in a model contains a list of parameters, a type for a possible return value and a list of non-successful terminations modelled by **exceptions**. **Exceptions** carry information and are defined in the same way as the **data type** *StructDef*. They contain a list of members for that information. An **attribute** is a shorthand notation for modelling operations used for getting and setting a named variable of a certain type. Also, **exceptions** can be added to **attributes**. In that case, it is distinguished between those **exceptions** which are possibly raised during a *set*-**operation** for that **attribute** and those being raised during a *get*-**operation**. The **metamodel** for **exceptions** is shown in Figure 8.

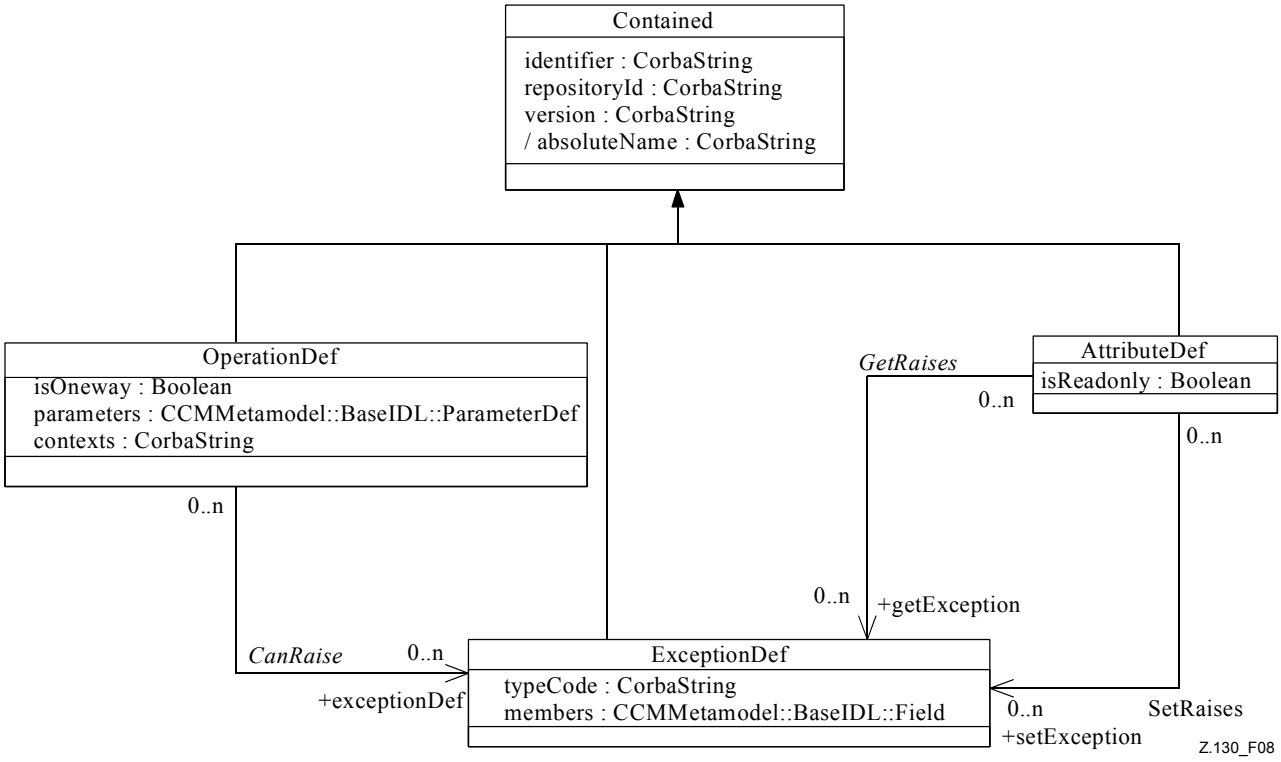


Figure 8/Z.130 – Exceptions

The metaclasses *OperationDef*, *AttributeDef*, *ParameterDef* and *ExceptionDef* are from the CORBA-IDL **metamodel** and are described in detail there. The **metamodel** for **attributes** and **operations** is shown below (see Figures 9 and 10).

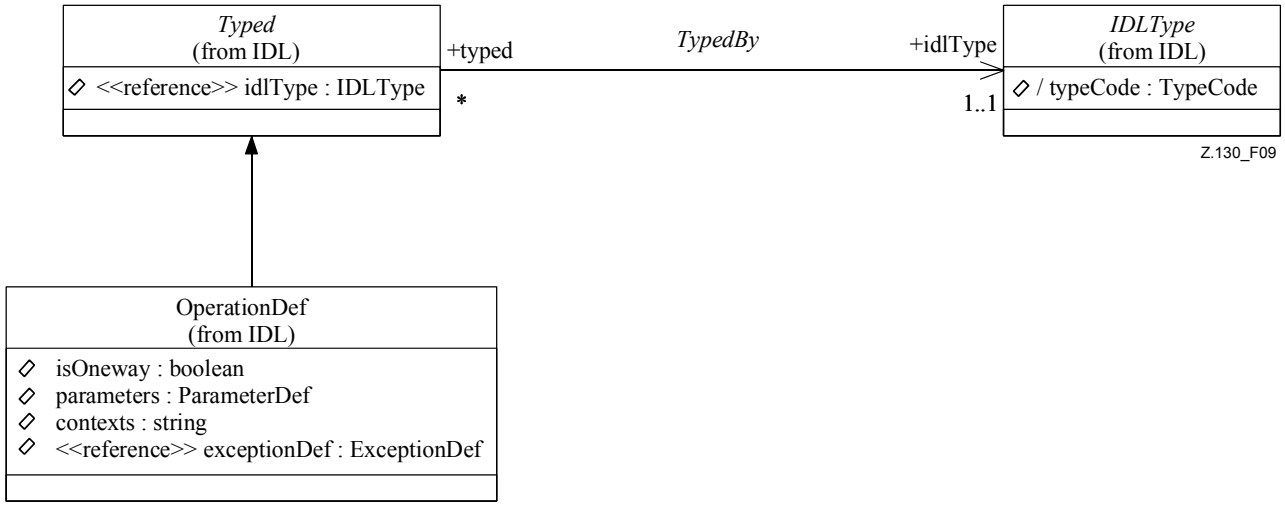


Figure 9/Z.130 – Operations

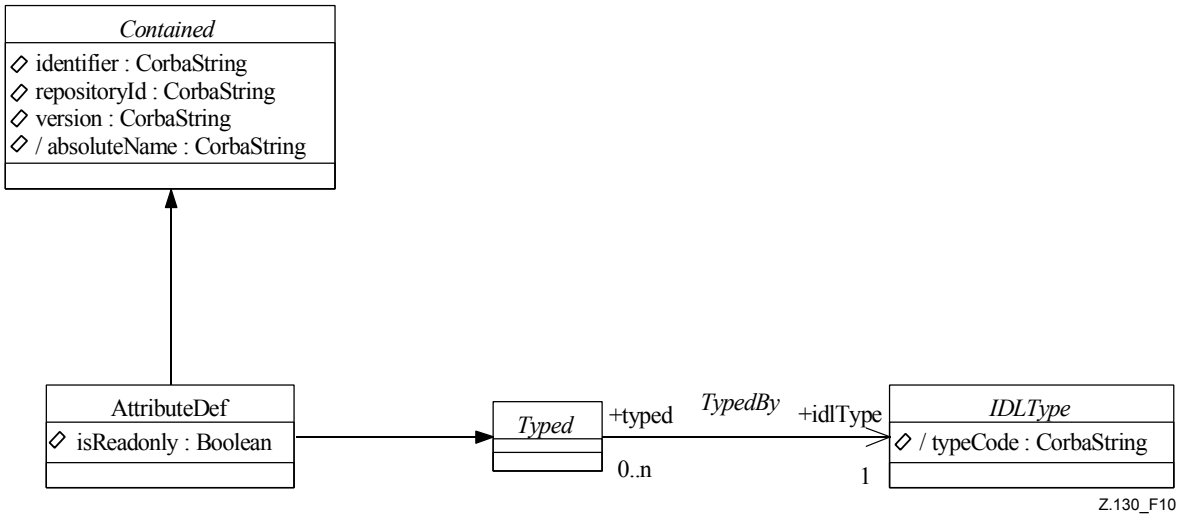


Figure 10/Z.130 – Attributes

Interface types are used to define signatures for possible **interactions** in a system. The concept of **interface type** is already known from OMG IDL, where it is named interface. In OMG IDL, interfaces only aggregate operational **interaction elements**. That means they are containers for **attributes** and **operations**. This is shown in Figure 11.

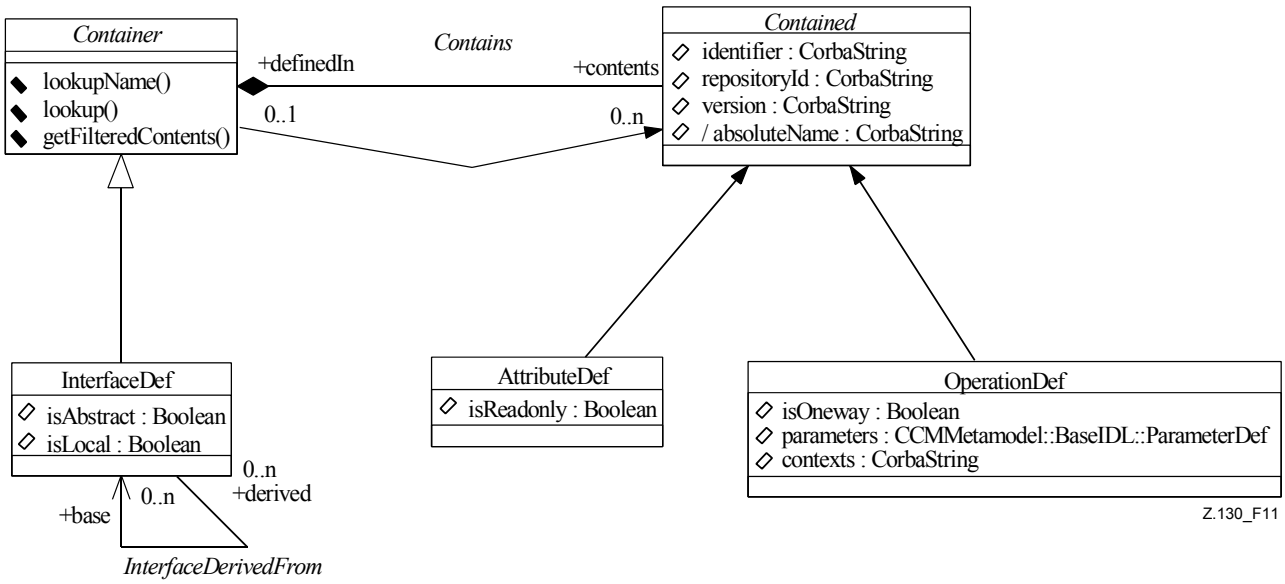


Figure 11/Z.130 – Operational interfaces

5.3.2 Signals and signal parameters

Metamodel

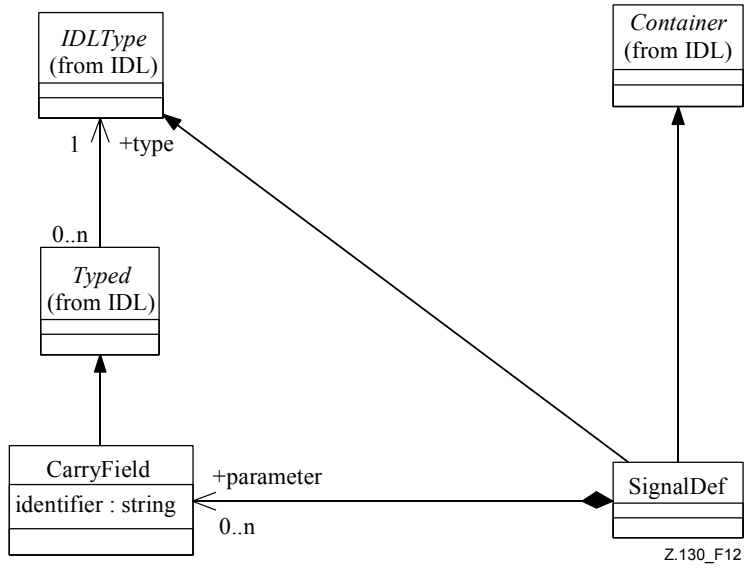


Figure 12/Z.130 – Signal and signal parameter

Semantics

To enhance the modelling concepts offered by CORBA-IDL and to include other interaction kinds into the modelling of distributed systems further modelling concepts are necessary (see Figure 12).

Signals are used to model **signal**-based interaction, that means the asynchronous decoupled message exchange between system entities. **Signals** carry information. **Signals** are modelled as instances of the metaclass *SignalDef*. The carried information (called **signal parameters**) is modelled as instances of *CarryField*, each referring to an instance of *ValueDef*, which is a special **data type** of CORBA-IDL. Each **signal parameter** is identifiable using a name in the context of a **signal** definition. The names have to be unique.

A **signal** definition prescribes the structure and the properties of the information which is carried in a concrete signal interaction between system entities. It is not yet associated to an **interface type**. **Signal** definitions can be reused in different **interface type** definitions. They play the same role as **data types** do for the definition of **operations**. They are building blocks for **signal**-based **interaction**.

Signal definitions in models can only occur in **name spaces** which are either modules or which is the global **name space** formed by the specification itself.

The IDL types which are used to specify the parameters of **signals** have to be instances of the type *value*.

5.3.3 Medium type, medium, media set

Metamodel

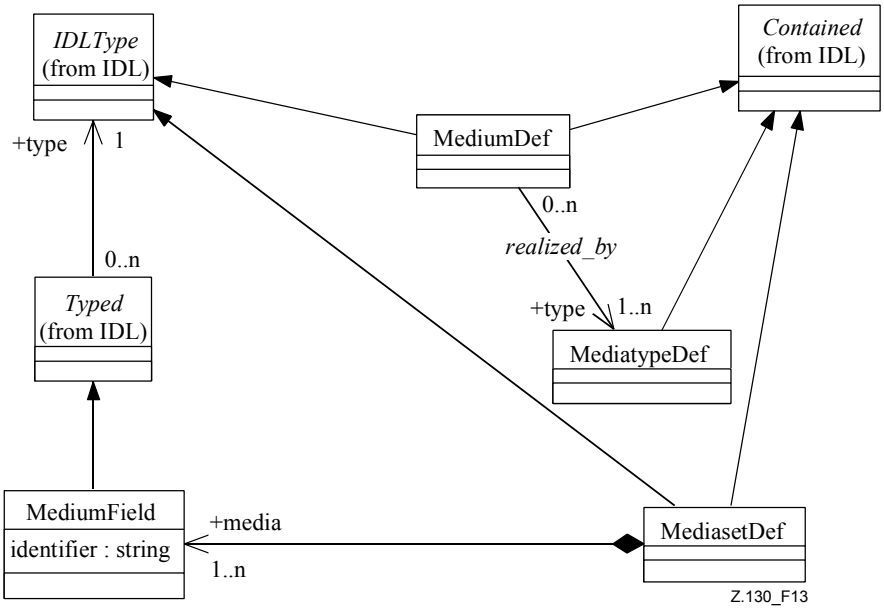


Figure 13/Z.130 – Medium, medium type, media set

Semantics

In addition to **operational** and **signal**-based interaction, the exchange of **continuous media** is also an important interaction kind in distributed systems. Modelling concepts for this interaction kind have to be provided. This is done in this Recommendation in exactly the same way as for **operational** and **signal** interaction. At first, the basic building blocks for the **interaction elements** have to be defined. These are **medium**, **media set** and **medium type** (see Figure 13).

The concept **media set** is used to model the **continuous media** interaction. In the **metamodel** this is provided by the definition of the class *MediasetDef*. Instances of this class aggregate instances of the class *MediumDef* in a named list where each element has the type *MediumField*. The concept of **medium** is used to model one atomic data flow between two entities. A **medium** has the meaning of multimedia information like films or audio sequences. The exchange requires the presence of coding, decoding and transmission formats, which are modelled by instances of the class *MediatypeDef*. A **medium** can be realized by one or more **media types**. **Media sets** are necessary to model the exchange of two different media which belong together (like video and audio data of a film) and where the interactions for both should have the same properties.

Definitions of **media**, **media sets** and **media types** in models can only occur in namespaces which are either modules or which is the global namespace formed by the specification itself.

5.3.4 Consume and produce

Metamodel

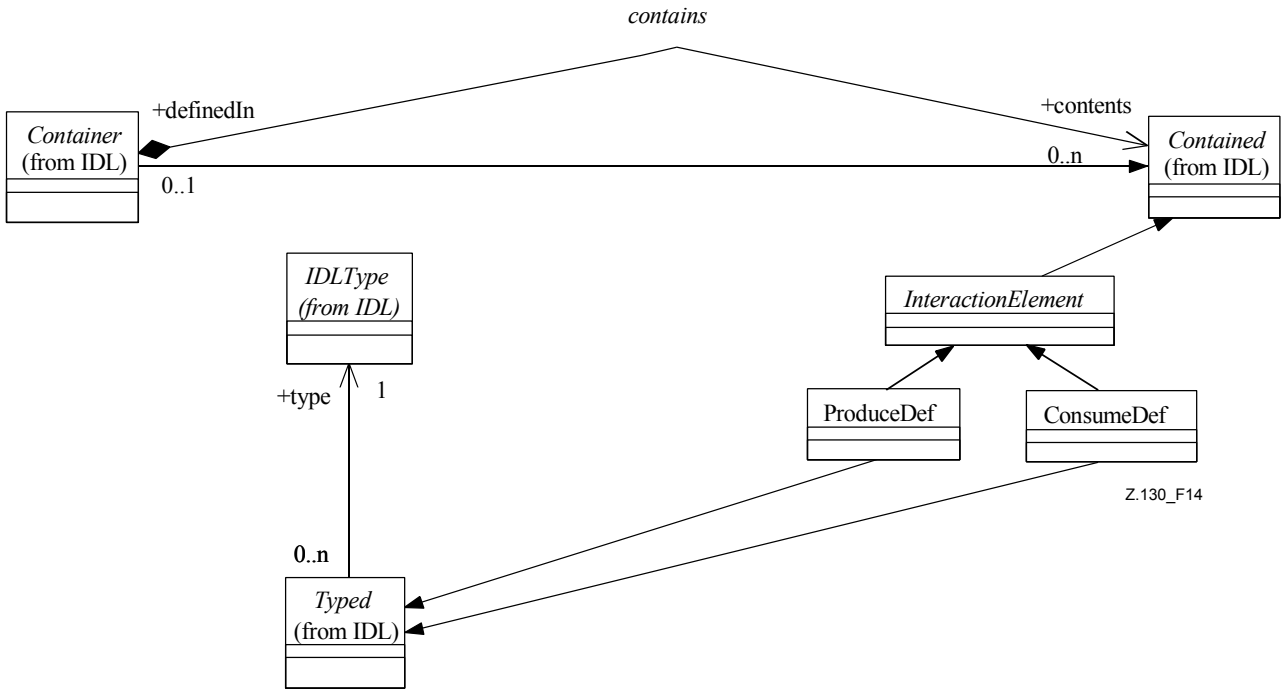


Figure 14/Z.130 – Consume and produce

Semantics

Signals will be exchanged between functional entities at **runtime** via their **interfaces**. For that reason, the **interface types** offer the necessary signatures. In the case of **signal** communication the signature contains the type (**signal** definition), a name for the **interaction element** and the indication whether the **signal** is produced or consumed via this interface. So, the concepts **consume** and **produce** are **interaction elements** and are used to model the consumption or production of **signals** (see Figure 14). They are elements of the **signal** interaction in the same sense as **operations** and **attributes** are the elements of the **operational** interaction. As every **interaction element**, **consume** and **produce** are identifiable elements. They are included in the **metamodel** with the definition of the metaclasses *ConsumeDef* and *ProduceDef*, which inherit from the abstract metaclass *InteractionElement*, which itself inherits from *Contained*. *ConsumeDef* and *ProduceDef* are subclasses of Typed. This inheritance is used to establish the relation to the *SignalDef*, which is addressed by **produce** or **consume**.

Note that *SignalDef* is a subclass of *IDLType*. It is required that the instance of *IDLType* associated to a **produce** or **consume** definition is a **signal**. The concept of **signal** is defined as a specialization of *IDLType* for that purpose.

5.3.5 Sink and Source

Metamodel

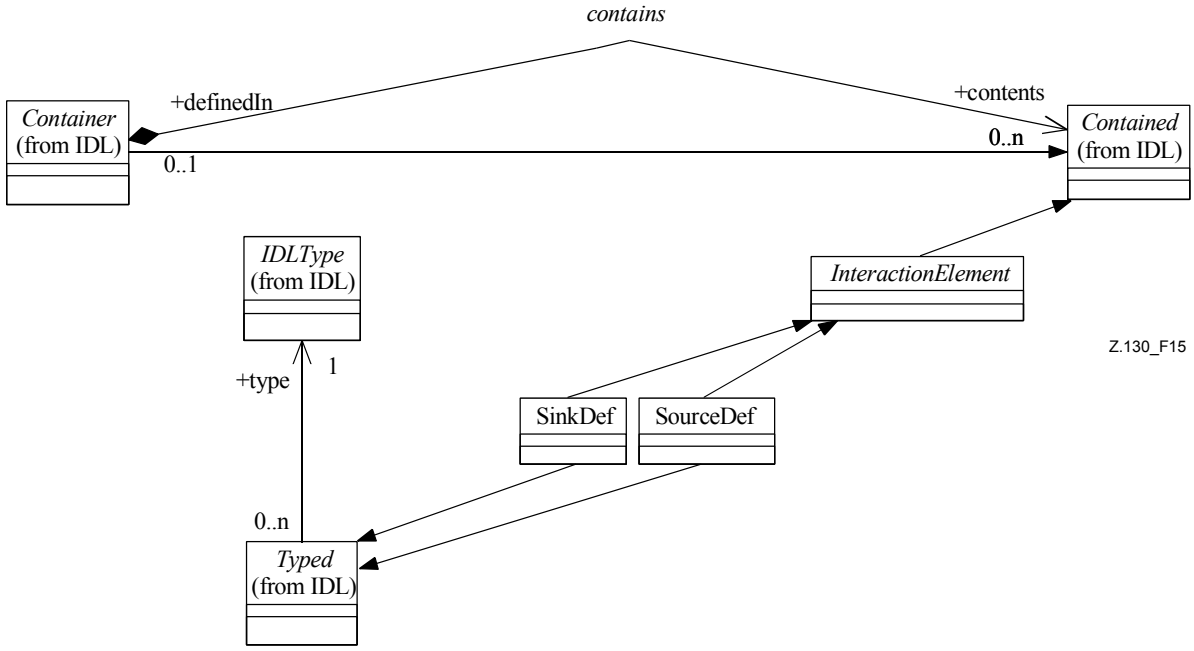


Figure 15/Z.130 – Sink and Source

Semantics

To complete the list of possible **interaction elements** which are used to specify **interface type** signatures, **interaction elements** for **continuous media** interactions have to be defined. Analogous to **signal** interactions, they are characterized by the information which is exchanged in case of an interaction at **runtime** (**media set**), an identifier and the direction of the communication, i.e., whether the **interface** is **sink** or **source** with respect to the interaction. Hence, the concepts **sink** and **source** are **interaction elements** to model the consumption or production of **media sets** (see Figure 15). They are elements of the **continuous media** interaction in the same sense as **operations** and **attributes** are the elements of the **operational** interaction. As every **interaction element**, **sink** and **source** are identifiable elements. They are included in the **metamodel** with the definition of the metaclasses *SinkDef* and *SourceDef*, which inherit from the abstract metaclass *InteractionElement*, which itself inherits from *Contained*. *SinkDef* and *SourceDef* are subclasses of *Typed*. This inheritance is used to establish the relation to the *MediasetDef*, which is addressed by the **sink** or the **source** concept.

Note that *MediasetDef* is a subclass of *IDLType*. The only concrete *IDLType* which is allowed to be associated to a **sink** or **source** is **medium type**.

5.3.6 Interface type

Metamodel

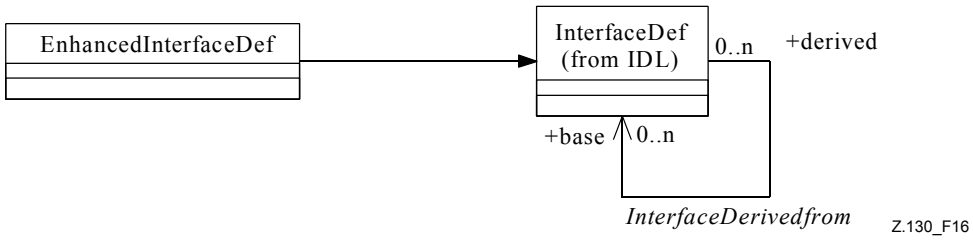


Figure 16/Z.130 – Interface type

Semantics

The concept **interface type** is used to specify a subset of potential **interactions** of **COs** of **CO types**. **Interface types** aggregate **interaction elements** of the interaction kinds **operational**, **signal** and **continuous media**. With this, the semantics of an **interface type** is extended compared to RM-ODP: **interface types** provide a common context for **interaction elements** of *different* interaction kinds. For clients, which have a reference to such an **interface**, it is possible to use all of the **interaction element**s independently of which interaction kind is used. It is a subject of the runtime environment to handle this aspect.

In the **metamodel**, **interface type** is an instance of class, named *EnhancedInterfaceDef* (see Figure 16). Since the **metamodel** of IDL already contains a meaning of aggregation of **operational interaction elements**, the class *EnhancedInterfaceDef* inherits from *InterfaceDef*. From this follows that the inheritance rules and constraints of *InterfaceDef* also apply to *EnhancedInterfaceDef*.

In addition, **interface types** are containers for the **interaction elements produce**, **consume**, **sink** and **source**.

5.3.7 CO types, supports and requires relation

Metamodel

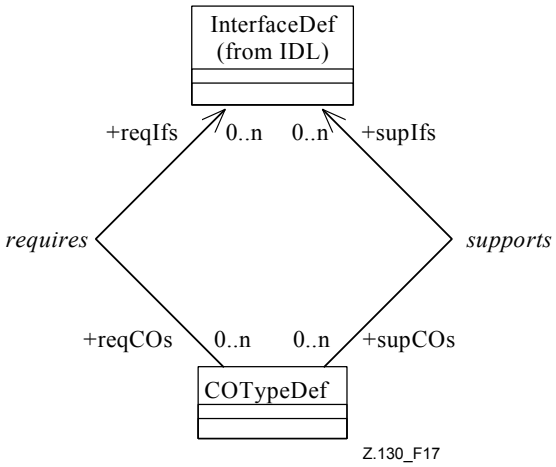


Figure 17/Z.130 – CO types, supports and requires

Semantics

The concept of **CO type** is used to specify the functional decomposition of a system. Instances of a **CO type** (**COs**) are autonomous interacting entities, which encapsulate state and behaviour. **COs** interact with their environment via well-defined **interfaces**. These **interfaces** are specified using the concept **interface type** described above.

A **CO type** may support (**supports**) or require (**requires**) an **interface type**. To support an **interface type** means that **COs** of that **CO type** provide **interfaces** of that **interface type**. To require an **interface type** means that **COs** of that **CO type** use **interfaces** of that **interface type**. A **CO type** is an instance of class *COTypeDef* in the **metamodel**. The labels supports and requires identify the associations between *COTypeDef* and *InterfaceDef* (see Figure 17).

In order to access **COs** at **runtime**, *COTypeDef* is derived from *InterfaceDef* (as shown in Figure 17). By doing so, instances can be configured using attributes which are defined by this **CO type**. It is important to note that it is only allowed for a **CO type** to contain **interaction elements** of the attribute kind. No other **interaction element** is permitted. Furthermore, the inheritance relations between **CO types** and **interface types** cannot be mixed, i.e., **CO types** can only inherit from **CO types** and **interface types** from **interface types**.

5.3.8 Provided and Used Port Definition

Metamodel

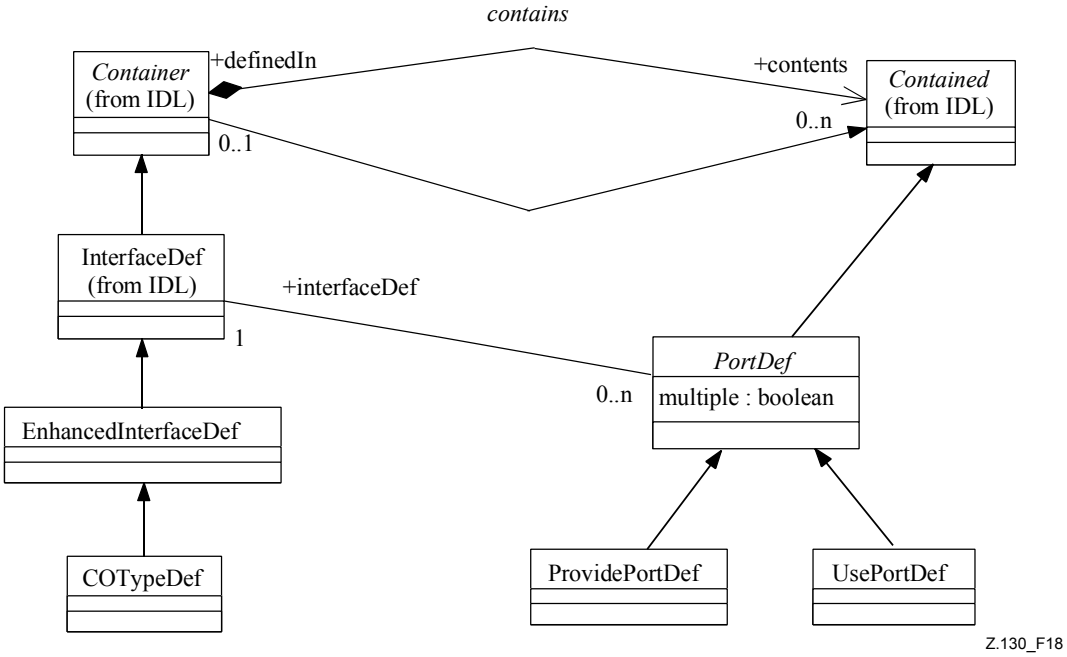


Figure 18/Z.130 – Provided and used port

Semantics

COs are the functional entities in a distributed system that is specified by using this Recommendation. They communicate via their supported and required **interfaces**. However, the configuration of distributed systems is always a problem, especially how to obtain and exchange **interface references**, which is a prerequisite for **interaction**. For this reason, this Recommendation introduces the concept of **port** as a named interaction point, at which either a reference of a supported **interface** of a **CO** can be obtained or a reference of a used **interface** can be registered at **runtime**.

The concepts **provided** and **used port** are used to model **ports** of a **CO type**, that are either used by the environment to obtain a reference to an **interface** (**provided port**) or to store a reference to an **interface** (**used port**) based on a name. With the concepts **supports** and **requires**, only the potential provision or usage of **interface types** in a context of a **CO type** can be expressed, but not the concrete mechanisms, how the environment of a **CO** gains access to these interaction contexts. The concepts **provided port** and **used port** are defined as instances of class *ProvidePortDef* and *UsePortDef*. Both classes inherit from the abstract class *PortDef*. The class *PortDef* inherits from *Contained*, meaning that a *COTypeDef* instance may contain **provided** and **used port** definitions. A **provided** and a **used port** definition are always associated to an **interface** definition (see Figure 18).

Port definitions are only allowed within **CO type** definitions. An **interface**, for which a **provided port** is defined, is automatically a supported **interface**. An **interface**, for which a **used port** is defined, is automatically a required **interface**.

5.4 Implementation concepts

5.4.1 Artefact and instantiation pattern

Metamodel

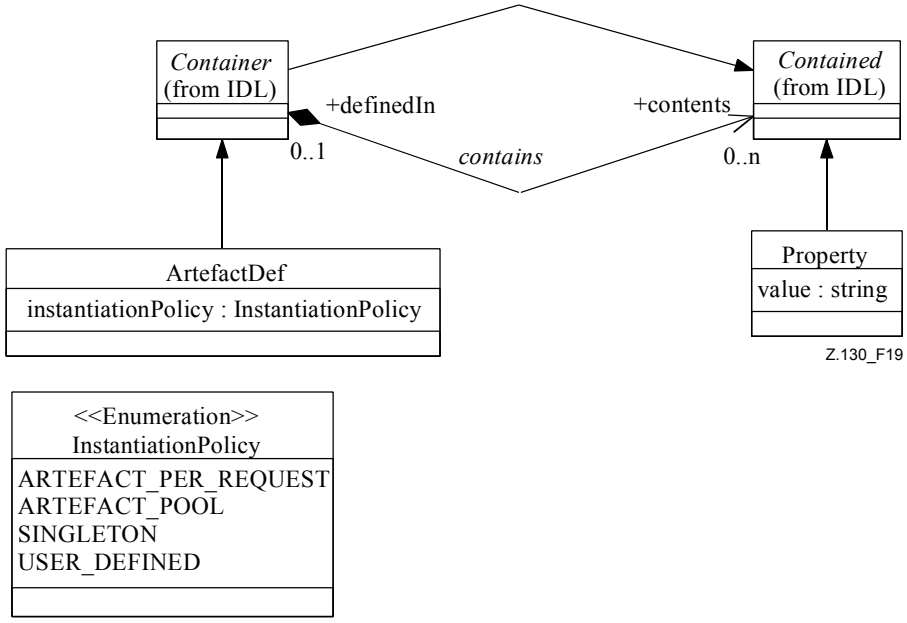


Figure 19/Z.130 – Artefact and instantiation pattern

Semantics

The concept **artefact** is used to describe a programming language context (such as a class of an object-oriented programming language) in a model. Instances of the concept **artefact** realize the behaviour of **COs**. They therefore provide the business logic of **CO types**. The relations between the **artefacts** and the behavioural parts of the **COs** are defined by associations between **artefacts** and **interaction elements** of **interface types**. The programming language contexts that are modelled by instances of the concept **artefact** will be instantiated at **runtime** to process, e.g., **operation** invocations, **signal** inputs or **continuous media** data. The policies to be used for the instantiation are specified by instances of the concept instantiation pattern. Allowed patterns are to be seen in Figure 19. If necessary, further patterns may be added. The separation of the concepts **artefact** and **CO type** provides full flexibility when designing a distributed application:

- The external view (how the environment can interact with a **CO**) is separated from the internal view (how the behaviour of a **CO** is provided).
- Different inheritance trees can be used for the external and internal views.
- Reuse of existing behaviour and existing **interface** definitions are possible and independent from each other.

The concept **artefact** is expressed in the **metamodel** by an instance of class named *ArtefactDef*. The concept instantiation pattern is modelled as an attribute of that class of type enumeration with the enumerators as depicted in Figure 19.

5.4.2 Implements relation

Metamodel

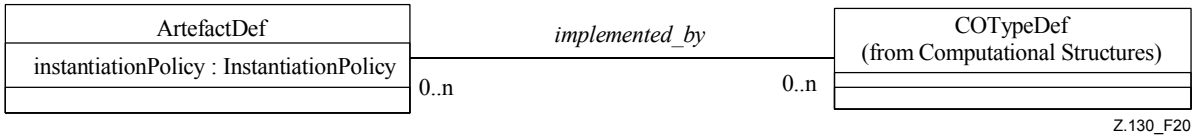


Figure 20/Z.130 – Implements relation

Semantics

Instances of the concept **artefact** describe the realization of the expected behaviour of **interaction elements** of **interface types** that are supported or required by **CO types**. As described above, the external and internal views on a **CO** are completely separated from each other. However, the relation between them has to be described in order to choose the right behaviour when a **CO** gets involved in a certain interaction. Therefore, the model describes which set of **artefacts** provide the behaviour for the **COs** of that **CO type**. This relation is defined by an association *implemented_by* between *COTypeDef* and *ArtefactDef* in the **metamodel** (see Figure 20).

5.4.3 Implementation element

Metamodel

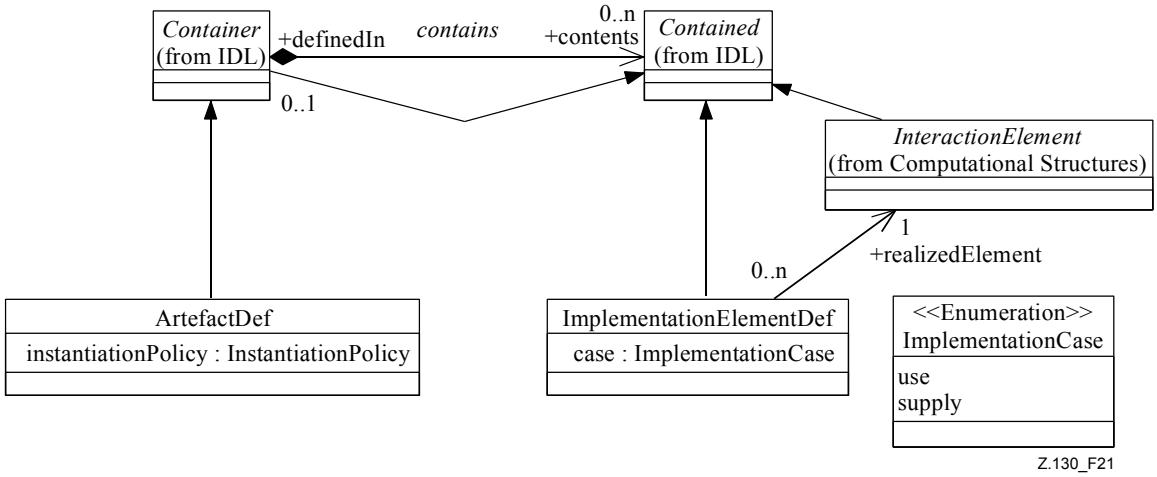


Figure 21/Z.130 – Implementation element

Semantics

In the context of an instance of **artefact**, the concept **implementation element** is used to denote that a behavioural part of an **artefact** (e.g., a method of a class that is modelled by an instance of the concept **artefact**) **realizes** a particular **interaction element** of an **interface type**. This concept is necessary to provide further details to the *implemented_by* relation as explained above. *implemented_by* specifies that an **artefact** contributes to the behaviour of a **CO** without saying which part of the **artefact** is responsible for what part of the **CO** behaviour. These details are provided with the **implementation elements** of the **artefact**. This information is necessary to associate the behaviour to an interaction at **runtime**.

The concept **implementation element** is specified as an instance of class with the name *ImplementationElementDef*. This class inherits from *Contained*; instances of *ImplementationElementDef* may be contained by instances of **artefacts** (see Figure 21). The *ImplementationCase* defines the implementation direction of an **implementation element**: Either the usage or the provision of the behaviour can be **realized** by an **implementation element** with respect to a certain **interaction element**.

5.5 Deployment concepts

5.5.1 Software component and component dependency

Metamodel

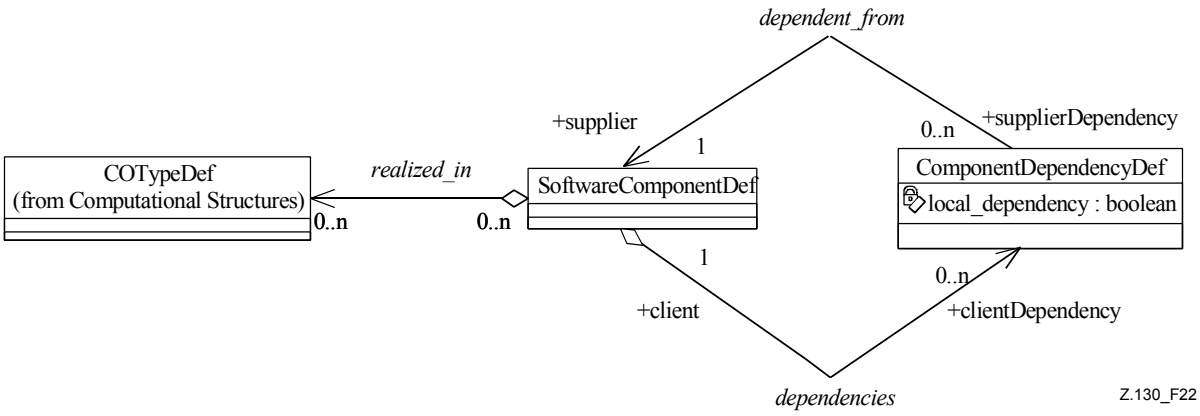


Figure 22/Z.130 – Software component and component dependency

Semantics

The concept of **software component** reflects actual software in the design model. It identifies an entity of **deployment** and allows further description by using properties. A **software component** may, but does not have to, **realize** an arbitrary number of **CO types**. Therefore, it contains sequences of instructions, which when executed on a node, incarnate **COs**, i.e., they provide behaviour, state and identity of **COs**. In the **metamodel**, the concept is introduced by the metaclass *SoftwareComponentDef*. To indicate which **CO types** are **realized** by a certain **software component**, there is the concept of **realize** relation, which is introduced in the **metamodel** by an association between the metaclass *COTypeDef* and *SoftwareComponentDef* (see Figure 22).

Software components may require other **software components** in order to be properly executed. To reflect this in the model, the metaclass *SoftwareDependencyDef* is defined. It contains the *local_dependency* attribute which states whether the required **software component** has to be locally available. A *SoftwareComponentDef* may contain an arbitrary number of *SoftwareDependencyDefs,* where each *SoftwareDependencyDef* has an association to another *SoftwareComponentDef* indicating the required software.

5.5.2 Assembly and initial configuration

Metamodel

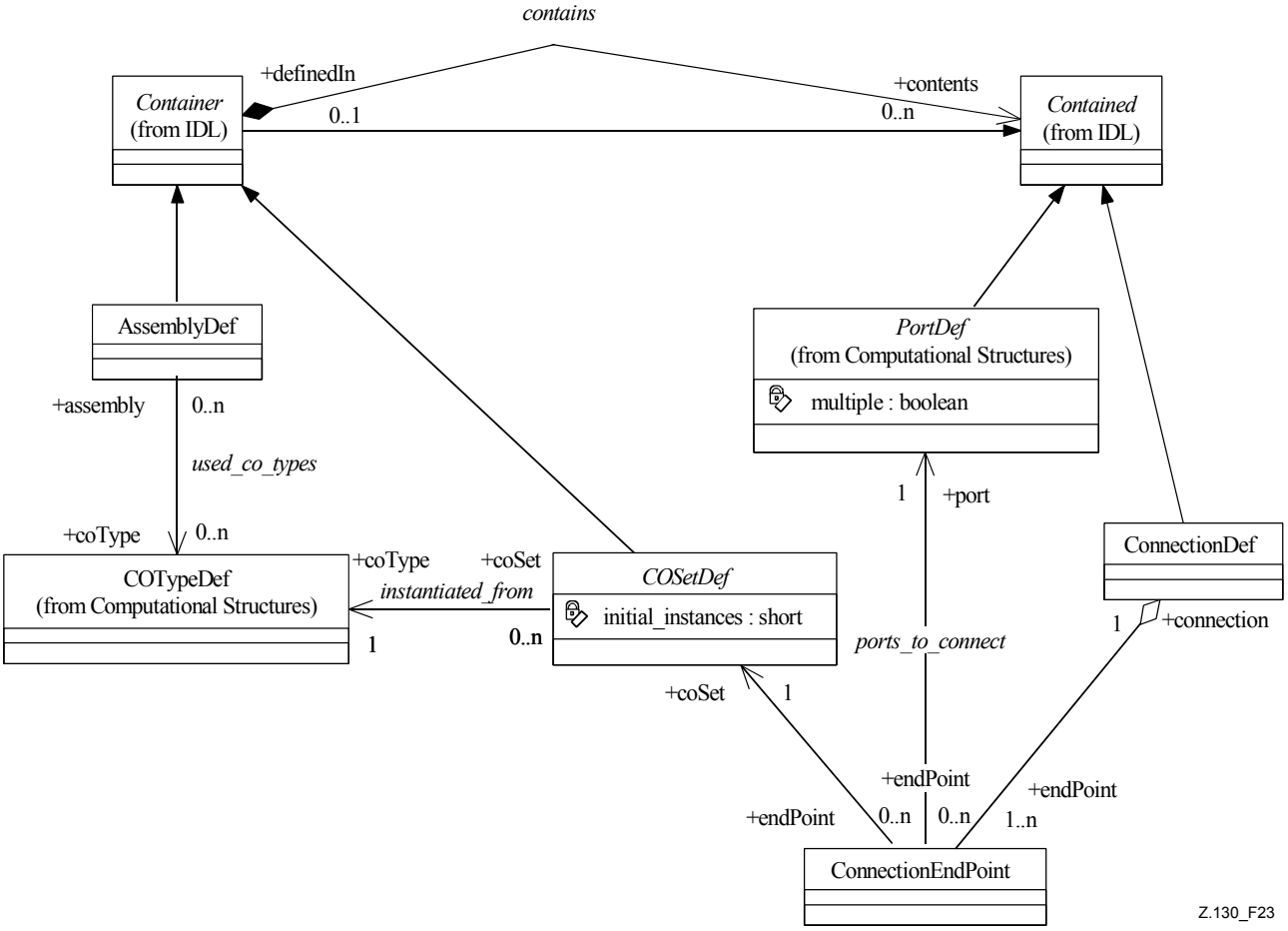


Figure 23/Z.130 – Assembly and initial configuration

Semantics

The concept of **assembly** is used to model software systems by specifying the **CO types** which are involved in the system and to model the **initial configuration** of the system. The **initial configuration** is the configuration which is established at the start of the execution time of the software system and consists of **initial COs** and their **initial connections**. In the **metamodel**, the concept of **assembly** is modelled by the metaclass *AssemblyDef*. The **CO types** are associated by the introduction of an association between the metaclasses *AssemblyDef* and *COTypeDef* (see Figure 23).

To model **initial COs**, the **metamodel** contains the metaclass *COSetDef*. A *COSetDef* defines the creation of an arbitrary number of instances of the associated **CO type**. The number is determined by the *initial_instances* attribute. A *COSetDef* is contained in an *AssemblyDef*.

To model **initial connections**, the **metamodel** contains the metaclass *ConnectionDef*. A **connection** is established between ports of the participating **COs** by the exchange of **interface references** of the **COs**. These references are obtained from a **CO** where the **CO type** has a **provided port** definition and is transferred to a **CO** whose type has a **used port** definition. In the **metamodel**, a *ConnectionDef* consists of a set of *ConnectionEndPoints*. A *ConnectionEndPoint* is associated with a *PortDef* of a *COTypeDef* and a *COSetDef*. Each **CO** of a *COSetDef* associated with a

ConnectionEndPoint is connected with each **CO** of each COSetDef associated with other *ConnectionEndPoints* aggregated to the same *ConnectionDef*.

5.5.3 Properties and constraints

Metamodel

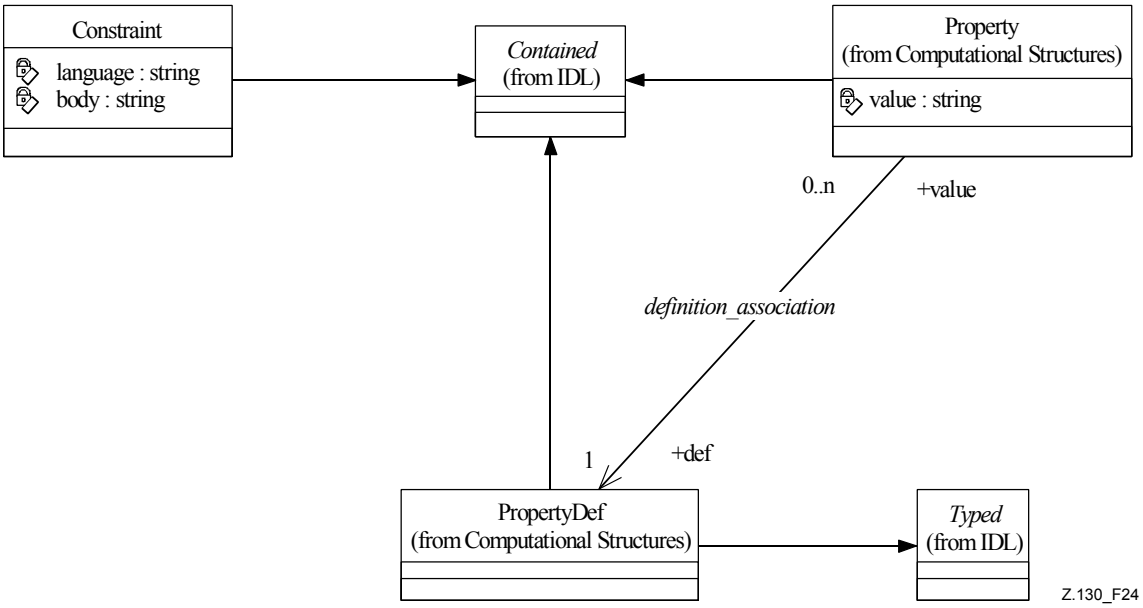


Figure 24/Z.130 – Properties and constraints

Semantics

The concept of **properties**, which can be attached to model elements is reflected in the **metamodel** by the metaclasses *PropertyDef* and *Property* (see Figure 24). The **metamodel** distinguishes between a property definition and a property value. A *Property* holds a value represented by the string attribute value while a *PropertyDef* holds the **data type** specification for the value, as it is inherited from the metaclass *Typed*. For example, a **CO type** may define the properties needed for its configuration, while a **CO** may define the appropriate property values.

The concept of **constraint** is reflected in the **metamodel** by the metaclass *Constraint*. It has two attributes. The *language* attribute determines the language in which the constraint is written, and should be used for evaluation and the *body* attribute, which contains the actual string representation of the constraint. The choice of a constraint language is left to the user and is not prescribed by this Recommendation. This way, any appropriate language can be chosen and the semantics of constraints are not defined here, but are left to the processing tools. A **CO** may define a set of constraints to express the permitted combinations of property values. An **assembly** may define collocation constraints on the running components. References property definitions or attributes are qualified by their names.

5.6 Target environment concepts

The specification of distribution and **deployment** is achieved by modelling actual target environments, logical target environments and by mapping logical entities to actual nodes of the environments. In some cases the model of the actual target environment can be retrieved automatically by a tool.

5.6.1 Target Environment, Node and NodeLink

Metamodel

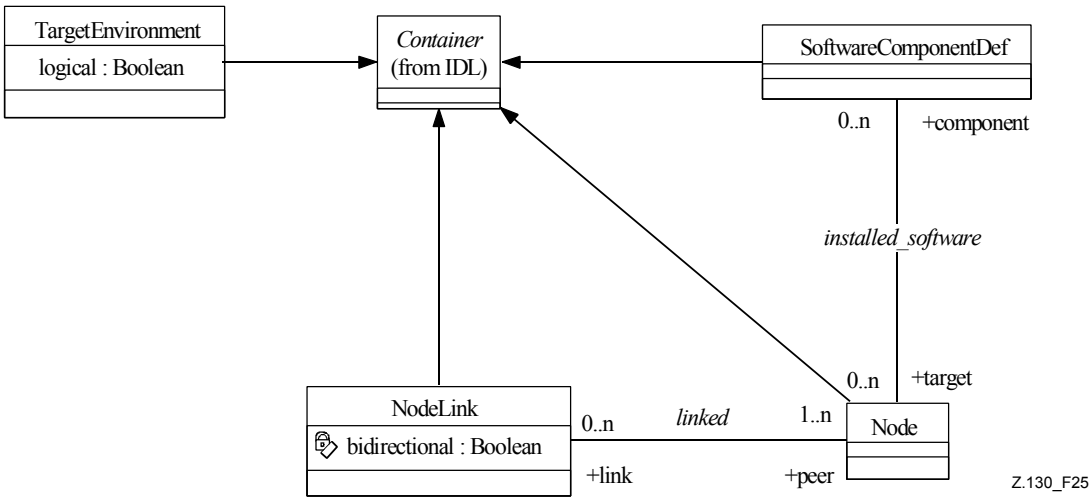


Figure 25/Z.130 – Target Environment, Node and NodeLink

Semantics

A target environment models a physical distributed runtime environment, e.g., telecommunication network consisting of nodes and links between nodes. In the **metamodel**, the metaclass *TargetEnvironment* is a container for *Node* and *NodeLink* (see Figure 25). The *logical* attribute in *TargetEnvironment* is used to indicate whether the model reflects an existing environment or a potential one.

The metaclass *Node* represents the concept of node, which means an element of the target environment that features at least a single or multiple processors, a memory unit and an operating system. Note that a given physical machine may host more than just one logical *Node*. A node refers to the installed software (**software components**, compilers, interpreters, etc.) and the installed hardware (represented as driver software). Properties such as the operating system or the processor are described as predefined properties listed below.

The concept of links between nodes is represented by the metaclass *NodeLink* as a physical link between two or more Nodes (a shared bus, for instance). The *bidirectional* Boolean attribute in *NodeLink* applies only in case the *NodeLink* is associated with exactly two *Nodes*. When *bidirectional* is false, the order of the two *Nodes* associated with *NodeLink* is interpreted as follows: the first *Node* instance corresponds to the source node, and the second *Node* instance represents the destination node. Therefore, the association *linked* is ordered.

A *Node* and a *NodeLink* are containers for *PropertyDef* and *Property*. This means predefined properties and user-defined properties can be attached directly to node instances and node link instances.

5.6.1.1 Node and NodeLinks predefined properties

There are some predefined properties for *Node* and *NodeLink* (see Table 1). For that purpose, the following implicit **data types** are introduced using IDL:

```
struct ProcessorType {
      string family;
      string type;
      integer frequency;
}
```

```
struct OSType {
      string name;
      string version;
}
```

Table 1/Z.130 – Predefined properties

Entity	Predefined Property name	Type	Description	Mandatory
Node	Processor	ProcessorType	the processor on the node	Yes
	Memory	Integer	the maximum amount of memory of the node (in kilobytes)	No
	OS	OSType	the node operating system identification	Yes
NodeLink	Bandwidth	Integer	The maximum bitrate of the link (in kilobytes per second)	No

5.6.2 InstallationMap

Metamodel

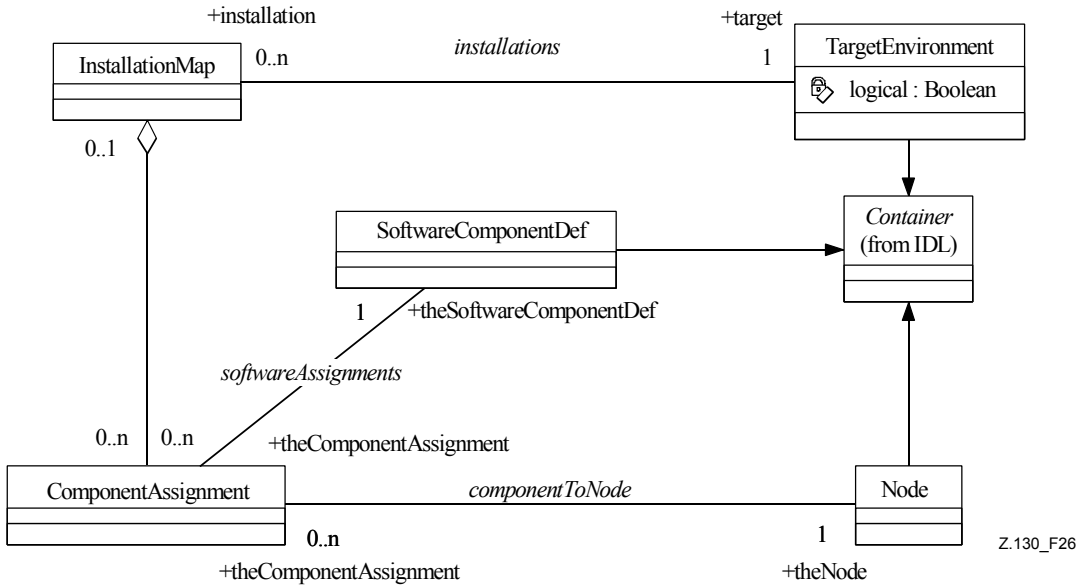


Figure 26/Z.130 – InstallationMap

Semantics

Once a *TargetEnvironment* is modelled, appropriate entities can be assigned to its nodes. There are two kinds of entities: software units, representing the needed software in a node, and **CO** sets, representing concrete instances of the **CO types**.

An installation map represents the way implementations are distributed on a target environment. It consists of a set of installation assignments, which each associates one **software component** with one node. An installation map refers to the nodes of a target environment. The representation of installation map is the metaclass *InstallationMap*. The representation of installation assignment is the metaclass *ComponentAssignment* (see Figure 26).

5.6.3 InstantiationMap

Metamodel

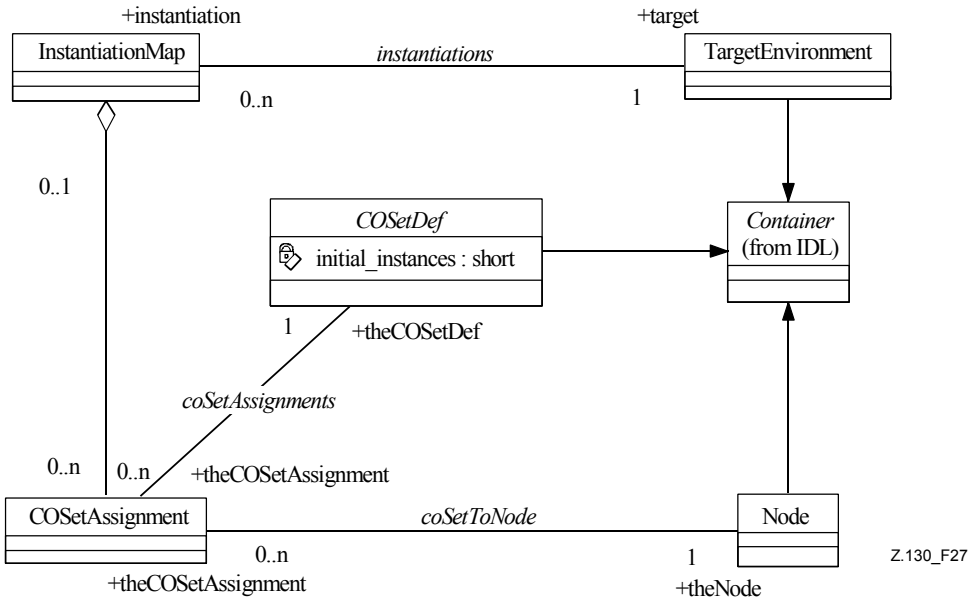


Figure 27/Z.130 – InstantiationMap

Semantics

An instantiation map represents the way concrete instances of **CO types** are distributed on a target environment. It consists of a set of instantiation assignments, which associate a set of **COs** with one node of the target environment. An instantiation map refers to **CO types** defined in the context of an **assembly** and to the nodes of a selected target environment. The metaclass representing the concept of instantiation map is *InstantiationMap*. The assignment of **COs** is represented by the metaclass *COSetAssignment* (see Figure 27).

5.6.4 Deployment plan

Metamodel

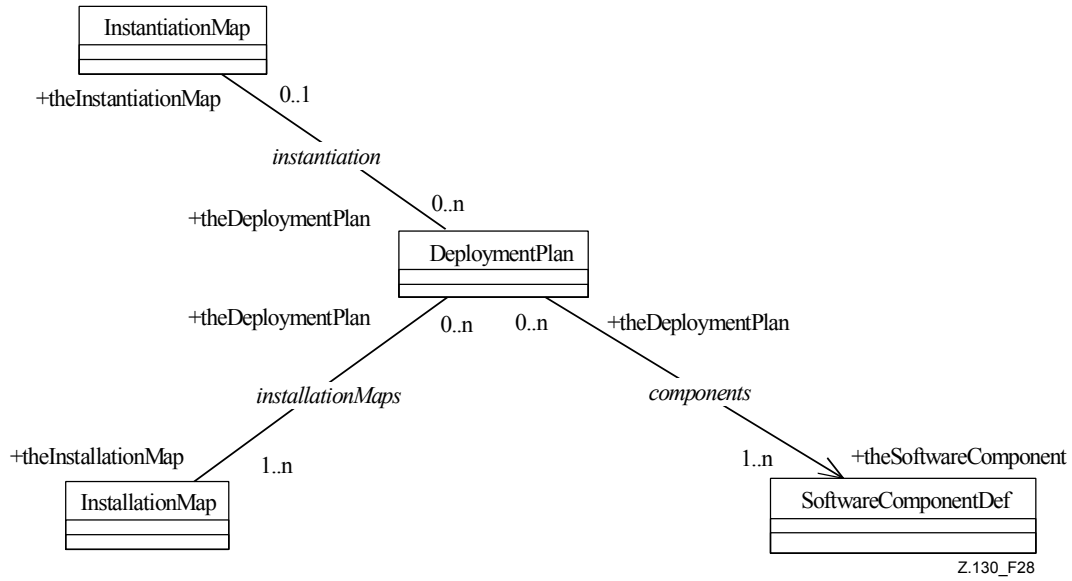


Figure 28/Z.130 – Deployment plan

Semantics

A deployment plan is defined by the selection of one or more **software components**, containing implementations for **CO types**, one or more installation maps, determining where to install **software components** and zero or one instantiation map, determining where to create **COs** (see Figure 28). The instantiation map and all installation maps of an eODL model have to refer to the same target environment and the same **assembly**.

6 Bibliography

[11] OMG Document formal/00-03-02, *OMG Unified Modeling Language Specification*, Version 1.3.

[12] W3C Recommendation (2000), Extensible Markup Language (XML) 1.0 (Second edition).

[13] OMG Document ad/01-02-29, *UML Profile for MOF*.

[14] OMG Document omg/ 00-11-05, *Model Driven Architecture*.

[15] IEEE Std. 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

[16] SZYPERSKI (C.): *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, ISBN: 0-201-17888-5.

Annex A

Syntax of eODL

A.1 Introduction

This annex provides the textual notation of eODL in EBNF style. The syntax for concepts originating from OMG CORBA IDL 2.4.2 is taken from [5].

A.2 Lexical conventions and grammar base

The definitions found in clause 3.1 (lexical conventions for IDL) and clause 3.4 (grammar of IDL) of OMG CORBA 2.4.2 specification document are applied in the subsequent clauses.

A.3 Computational view

A.3.1 Name spaces, data types, exceptions, operations and attributes

The metamodel is based on the CORBA-IDL data type system; the language eODL is based on CORBA-IDL. These foundations provide a canonical mapping of metamodel data types, **name spaces** and **exceptions** to eODL. **Operational interaction elements** of metamodel compliant models are mapped onto CORBA-IDL **operations** and attributes in the same canonical way.

A.3.2 Signals and carried values

One of the extensions of the metamodel compared to the conceptual foundation of ITU-ODL is the introduction of **signal interaction elements**. These **interaction elements** are based on the definition of **signals**. **Signals** in eODL are defined based on the following grammar:

```
<signal_dcl>      ::=   "signal" <identifier> "{" <member_list> "}"
<member_list>     ::=   <member>+
<member>          ::=   <type_spec> <declarators> ";"
```

A.3.3 Medium type, medium and media set

The semantics of stream interactions is left open in the conceptual foundation of ITU-ODL. The metamodel precisely defines the semantics of **continuous media** interactions that replaces stream interaction in ITU-ODL. The representation of the concepts **medium**, **medium type** and **media set** defined within the metamodel in eODL is given by the following grammar:

```
<mediaset_dcl>    ::=   "mediaset" <identifier> "{" <member_list> "}"
<mediatype_dcl>   ::=   "mediatype" <identifier>
<medium_dcl>      ::=   "medium" <identifier>  "(" <scoped_name> { "," <scoped_name> }* ")"
```

A.3.4 Interface types and interaction elements

An **interface type** in the metamodel combines **interaction elements** of different interaction kinds within a single interaction context. For the syntactical representation of this concept, the interface construct is extended by the notion of **source**, **sink**, **produce** and **consume interaction elements** in addition to **operational interaction elements**.

```
<interface>       ::=   <interface_dcl>
                  |     <forward_dcl>
<interface_dcl>   ::=   <interface_header> "{" <interface_body> "}"
<forward_dcl>     ::=   [ "abstract" ] "interface" <identifier>
<interface_header>::=   [ "abstract" ] "interface"  <identifier> [ <interface_inheritance_spec> ]
<interface_body>  ::=   <export> *
```

```
<export>                ::=  <type_dcl> ";"
                        |    <const_dcl> ";"
                        |    <except_dcl> ";"
                        |    <attr_dcl> ";"
                        |    <op_dcl> ";"
                        |     <produce_dcl> ";"
                        |    <consume_dcl> ";"
                        |     <source_dcl> ";"
                        |    <sink_dcl> ";"
<produce_dcl>           ::=  "produce" <scoped_name> <identifier>
<consume_dcl>           ::=  "consume" <scoped_name> <identifier>
<source_dcl>            ::=  "source" <scoped_name> <identifier>
<sink_dcl>              ::=  "sink" <scoped_name> <identifier>
```

A.3.5 Computational object type

ITU-ODL already allows the notion of **computational object type** (**CO type**). Through the precise definition of the configuration view onto a **CO type**, the concept of the initial interface is deprecated. The grammar for **CO types** from ITU-ODL is changed in eODL, and is defined by the following rules:

```
<object_template>           ::=  <object_template_header> "{" <object_template_export> "}"
<object_template_header>    ::=  "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
<object_template_export>    ::=  <object_export>*
<object_export>             ::=  <export>
                            |    <reqrd_interf_templates> ";"
                            |    <suptd_interf_templates> ";"
                            |    <use_dcl> ";"
                            |    <provide_dcl> ";"
                            |    <implements_dcl> ";"
                            |    <state_def_dcl> ";"
                            |    <constraint_dcl> ";"
                            |    <property_list>
<reqrd_interf_templates>    ::=  "requires" <scoped_name> { "," <scoped_name> }*
<supd_interf_templates>     ::=  "supports" <scoped_name> { "," <scoped_name> }*
```

A.3.6 Property

A property is used to define available or needed properties of model elements. The property notion is used for target environment and software unit definition.

```
<property_list>             ::=  { <property_dcl> ";" }*
<property_dcl>              ::=  "property" <property_name> "=" <property_value>
<property_name>             ::=  <identifier>
<property_value>            ::=  <simple_property_value>
                            |    <structured_property_value>
                            |    <sequence_property_value>
```

```
<simple_property_value>      ::=   <string_literal>
                             |     <integer_literal>
                             |     <boolean_literal>
<structured_property_value>  ::=   "{"  <property_assign>*  "}"
<sequence_property_value >   ::=   "["  <property_value>*  "]"
<property_assign>            ::=   <property_name> "="  <property_value> ";"
<constraint_dcl>             ::=   "constraint" <identifier> "{" <constraint_body> "}"
<constraint_body>            ::=   "language" "=" <string_literal> "body" "=" <string_literal> ";"
```

A.3.7 External type

An external type is used by an identifier to refer to an externally provided data type.

```
<extern_type>                ::=   "extern" "type" <identifier> <string_literal>
```

A.4 Configuration view

A.4.1 Ports

The main concept of the configuration view onto **COs** is the **port** concept. The notation for single and dynamic **ports** is defined by the following rules:

```
<object_export>              ::=   <export>
                             |     <reqrd_interf_templates> ";"
                             |     <suptd_interf_templates> ";"
                             |     <use_dcl> ";"
                             |     <provide_dcl> ";"
                             |     <implements_dcl> ";"
                             |     <state_def_dcl> ";"
                             |     <constraint_dcl> ";"
                             |     <property_list>
<use_dcl>                    ::=   "use" [ "multiple" ] <scoped_name> <identifier>
<provide_dcl>                ::=   "provide" [ "multiple" ] <scoped_name> <identifier>
```

A.5 Implementation view

A.5.1 Artefacts and implementation elements

Artefacts abstract from concrete programming language constructs that implement the behaviour of **COs**. The representation of **artefacts** and **implementation elements** in eODL is given by the following rules:

```
<artefact>                   ::=   <artefact_dcl>
                             |     <artefact_forward_dcl>
<artefact_forward_dcl>       ::=   "artefact"  <identifier>
<artefact_dcl>               ::=   <artefact_header> "{" <artefact_body> "}"
<artefact_header>            ::=   "artefact" <identifier> [ <artefact_inheritance_spec> ]
<artefact_inheritance_spec>  ::=   ":" <scoped_name> { "," <scoped_name> }*
<artefact_body>              ::=   <impl_elem_dcl>*
<impl_elem_dcl>              ::=   <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl>              ::=   "supply" | "use"
```

A.5.2 Implements Relations and Instantiation Policies

The **implemented_by** relation defines which **artefact** is used for the realization of a **CO type** behaviour. This relation is defined in eODL in the context of **CO type** definitions:

```
<object_export>            ::=  <export>
                           |    <reqrd_interf_templates> ";"
                           |    <suptd_interf_templates> ";"
                           |    <use_dcl> ";"
                           |    <provide_dcl> ";"
                           |    <implements_dcl> ";"
                           |    <state_def_dcl> ";"
                           |    <constraint_dcl> ";"
                           |    <property_list>
<implements_dcl>           ::=  "implemented" "by" <artefact_with_policy>
                                { "," <artefact_with_policy> }*
<artefact_with_policy>     ::=  <scoped_name> [ "with" <instantiation_policy_dcl> ]
<instantiation_policy_dcl> ::=  "ArtefactPool"
                           |    "ArtefactPerRequest"
                           |    "Singleton"
                           |    "UserDefined"
```

A.5.3 State types

In many implementation cases an **artefact** realization needs to access state information of the **COs** that it implements. State information of **COs** is defined in **CO types** given by the following rules:

```
<object_export>            ::=  <export>
                           |    <reqrd_interf_templates> ";"
                           |    <suptd_interf_templates> ";"
                           |    <use_dcl> ";"
                           |    <provide_dcl> ";"
                           |    <implements_dcl> ";"
                           |    <state_def_dcl> ";"
                           |    <constraint_dcl> ";"
                           |    <property_list>
<state_def_dcl>            ::=  "state" <scoped_name>
                                [ "provided" "to" "(" <provided_to_dcl> ")" ]
<provided_to_dcl>          ::=  <scoped_name> { "," <scoped_name> }*
```

A.6 Deployment view

A.6.1 Softwarecomponent

A concrete implementation of a **CO type** is represented by a Software Unit Definition. Within a Software Unit Definition multiple **CO types** can be **realized**. Software Units may depend on other Software Units and/or require other properties and services from the final execution environment.

```
<softwarecomponent_dcl>     ::=     <softwarecomponent_header>
                                    "{" <softwarecomponent_body> "}"
<softwarecomponent_header>          ::=     "softwarecomponent"  <identifier>
                                            "realizes"  <cotype_identifier_list>
<cotype_identifier_list>    ::=     <cotype_identifier> { ","  <cotype_identifier> }*
<softwarecomponent_body>    ::=     <softwarecomponent_stmt>*
<softwarecomponent_stmt>    ::=     "dependent"  "{"  <softwarecomponent_list>  "}" ";"
                            |       "requires"  "{"  <property_list>  "}"  ";"
<softwarecomponent_list>    ::=     <softwarecomponent_identifier>
                                    {  ","  <softwarecomponent_identifier>  }*
<softwarecomponent_identifier>      ::=     <scoped_name>
```

A.6.2 Assembly

An **assembly** describes a set of interconnected components, and has no relation to a concrete distribution in a distributed processing environment.

A.6.2.1 Assembly definition

The **assembly** definition contains definitions for all instance set definitions belonging to the **assembly** and **connection** definitions for instances in the **assembly**.

```
<assembly_dcl>          ::=     <assembly_header> "{"  <assembly_body>  "}"
<assembly_header>       ::=     "assembly"  <identifier>
<assembly_body>         ::=     <assembly_stmt>*
<assembly_stmt>         ::=     <instance_set_dcl> ";"
                        |       <connect_dcl> ";"
                        |       <constraint_dcl> ";"
                        |       <property_list>
```

A.6.2.2 Instance set definition

The instance set definition describes a non-empty set of instances of a **CO type**.

```
<instance_set_dcl>      ::=     <identifier> [ "(" <integer_literal> ")" ] ":" <cotype_identifier>
<cotype_identifier>     ::=     <scoped_name>
```

A.6.2.3 Connection definition

Connections between instances and instance sets are expressed with the connection definition. Here, **ports** according to the **CO type** definition are interconnected, where one **port** acts as **source** and the other as **sink**.

```
<connect_dcl>           ::=     "connect"  [ <identifier> ]  "{"  <connection_list>  "}"
<connection_list>       ::=     { <connection> ";" } +
<connection>            ::=     <instance_set_identifier>  "."  <port_identifier> "="
                                <instance_set_identifier>  "."  <port_identifier>
<instance_set_identifier>       ::=     <scoped_name>
<port_identifier>       ::=     <scoped_name>
```

A.6.3 Installation map definition

An installation map definition describes an assignment of **CO types** to nodes of a target environment. In a later installation action, one can refer to the installation map definition and trigger the installation of software units to **nodes**.

```
<installation_map_dcl>      ::=  <installation_map_header> "{" <installation_map_body> "}"
<installation_map_header>   ::=  "installation" <identifier>
                                 "uses" "environment" <environment_identifier>
<installation_map_body>     ::=  <install_stmt> *
<install_stmt>              ::=  <softwarecomponent_identifier> "->" <node_identifier> ";"
<environment_identifier>    ::=  <scoped_name>
```

A.6.4 Instantiation map definition

An instantiation map definition describes a concrete assignment of instance sets to **nodes** of the specified target environment and **assembly**.

```
<instantiation_map_dcl>      ::=  <instantiation_map_header> "{" <instantiation_map_body> "}"
<instantiation_map_header>   ::=  "instantiation" <identifier> <instantiation_map_header_env>
                                  <instantiation_map_header_ass>
<instantiation_map_header_env>  ::=  "uses" "environment" <environment_identifier>
<instantiation_map_header_ass>  ::=  "uses" "assembly" <assembly_identifier>
<assembly_identifier>        ::=  <scoped_name>
<instantiation_map_body>     ::=  <assign_instance_stmt>*
<assign_instance_stmt>       ::=  <instance_set_identifier_list> "->" <node_identifier> ";"
<instance_set_identifier_list> ::=  <instance_set_identifier> { "," <instance_set_identifier> }*
```

A.6.5 Deployment action

A deployment action is a sequence of installation and instantiation actions to be executed during **deployment**.

```
<deployment_action>          ::=  "deploy" "{" <deployment_body> "}" ";"
<deployment_body>            ::=  "install" "{" <install_list> "}" ";"
                                  "instantiate" "{" <instantiation_list> "}" ";"
```

A.6.5.1 Installation action

An installation action specifies an installation of a software unit onto an execution node in a target environment.

```
<install_list>               ::=  <install_member>*
<install_member>             ::=  <installation_map_identifier> ";"
                             |    <qualified_install_stmt>
<qualified_install_stmt>     ::=  <softwarecomponent_identifier> "->"
                                  <environment_identifier> "." <node_identifier> ";"
<installation_map_identifier> ::=  <scoped_name>
```

A.6.5.2 Instantiation action

An instantiation action specifies an instantiation of a **CO** set on an execution node in a target environment.

```
<instantiation_list>         ::=  <instantiation_member>*
```

```
<instantiation_member>          ::=  <instantiation_map_identifier> ";"
                                |    <qualified_assign_instance_stmt> ";"
<instantiation_map_identifier>  ::=  <identifier>
<qualified_assign_instance_stmt> ::= <assembly_identifier> "."  <instance_set_identifier>
                                     "->" <environment_identifier> "." <node_identifier>
```

A.7 Target environment

A target environment serves as a possible execution environment for **assemblies**. It reflects structure and properties of that environment. An eODL textual syntax may contain more than one target environment specification.

A.7.1 Environment definition

The environment definition describes a possible execution environment in terms of available **nodes** and communication **links**.

```
<environment_dcl>          ::=  <environment_header> "{" <environment_body> "}"
<environment_header>       ::=  "environment" <identifier>
<environment_body>         ::=  <environment_stmt>+
<environment_stmt>         ::=  <node_dcl> ";"
                           |    <link_dcl> ";"
```

A.7.2 Node definition

A node definition reflects an identifiable execution **node** in the target environment, which can be target for installation of **CO types** and instantiation of instance sets. Properties in the node definition characterize facilities of the execution node.

```
<node_dcl>                 ::=  "node" <identifier> "{" <property_list> "}"
```

A.7.3 Link definition

Communication links between execution nodes in the target environment are represented as link definitions. Properties in the link definition are related to the characteristics and the kind of the communication link.

```
<link_dcl>                 ::=  <link_header>  "{" <link_body> "}"
<link_header>              ::=  "link" <identifier>
<link_body>                ::=  "node" <node_list> ";" <property_list> ";"
<node_list>                ::=  <node_identifier> { "," <node_identifier> }*
<node_identifier>          ::=  <scoped_name>
```

A.8 Syntax of eODL

This clause gives the complete set of the production rules for eODL. It also includes all rules inherited from the base syntax of OMG IDL 2.4.2.

```
<specification>            ::=  <definition>+ [ <deployment_action> ]
<definition>       ::=      <type_dcl> ";"
                   |        <const_dcl> ";"
                   |        <except_dcl> ";"
                   |        <interface> ";"
                   |        <object_template> ";"
                   |        <artefact> ";"
```

```
                    |       <module> ";"
                    |        <value> ";"
                    |       <signal_dcl> ";"
                    |       <mediaset_dcl> ";"
                    |       <mediatype_dcl> ";"
                    |       <medium_dcl> ";"
                    |       <assembly_dcl> ";"
                    |       <softwarecomponent_dcl> ";"
                    |       <environment_dcl> ";"
                    |       <installation_map_dcl> ";"
                    |       <instantiation_map_dcl> ";"
<module>            ::=     "module" <identifier> "{" <definition> + "}"
<object_template>           ::=     <object_template_header> "{" <object_template_export> "}"
<object_template_header>    ::=     "CO" <identifier> [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=     ":" <scoped_name> { "," <scoped_name> }*
<object_template_export>    ::=     <object_export>*
<object_export>     ::=     <export>
                    |       <reqrd_interf_templates> ";"
                    |       <suptd_interf_templates> ";"
                    |       <use_dcl> ";"
                    |       <provide_dcl> ";"
                    |       <implements_dcl> ";"
                    |       <state_def_dcl> ";"
                    |       <constraint_dcl> ";"
                    |       <property_list>

<reqrd_interf_templates>    ::=     "requires" <scoped_name> { "," <scoped_name> }*
<supd_interf_templates>     ::=     "supports" <scoped_name> { "," <scoped_name> }*
<use_dcl>       ::=  "use" [ "multiple" ] <scoped_name> <identifier>
<provide_dcl>   ::=  "provide" [ "multiple" ] <scoped_name> <identifier>
<artefact>      ::=  <artefact_dcl>
                |    <artefact_forward_dcl>
<artefact_forward_dcl>      ::=     "artefact"  <identifier>
<artefact_dcl>  ::=         <artefact_header> "{" <artefact_body> "}"
<artefact_header>           ::=     "artefact" <identifier> [ <artefact_inheritance_spec> ]
<artefact_inheritance_spec>         ::=     ":" <scoped_name> { "," <scoped_name> }*
<artefact_body>     ::=     <impl_elem_dcl>*
<impl_elem_dcl>     ::=     <identifier> "implements" <impl_case_dcl> <scoped_name> ";"
<impl_case_dcl>     ::=     "supply" | "use"
<implements_dcl>    ::=     "implemented" "by" <artefact_with_policy>
                            { "," <artefact_with_policy> }*
<artefact_with_policy>      ::=     <scoped_name> [ "with" <instantiation_policy_dcl> ]
```

```
<instantiation_policy_dcl>      ::=     "ArtefactPool"
                                |       "ArtefactPerRequest"
                                |       "Singleton"
                                |       "UserDefined"
<state_def_dcl>         ::=     "state" <scoped_name> [ "provided" "to" "(" <provided_to_dcl> ")" ]
<provided_to_dcl>       ::=     <scoped_name> { "," <scoped_name> }*
<interface>         ::= <interface_dcl>
                    |   <forward_dcl>
<interface_dcl>         ::=     <interface_header> "{" <interface_body> "}"
<forward_dcl>       ::= [ "abstract" ] "interface" <identifier>
<interface_header>  ::=     [ "abstract" ] "interface"  <identifier> [ <interface_inheritance_spec> ]
<interface_body>    ::=     <export> *
<export>        ::= <type_dcl> ";"
                |   <const_dcl> ";"
                |   <except_dcl> ";"`
                |   <attr_dcl> ";"
                |   <op_dcl> ";"
                |   <produce_dcl> ";"
                |   <consume_dcl> ";"
                |   <source_dcl> ";"
                |   <sink_dcl> ";"
<produce_dcl>       ::=     "produce" <scoped_name> <identifier>
<consume_dcl>       ::=     "consume" <scoped_name> <identifier>
<source_dcl>    ::= "source" <scoped_name> <identifier>
<sink_dcl>      ::= "sink" <scoped_name> <identifier>
<interface_inheritance_spec> ::=    ":" <interface_name> { "," <interface_name> } *
<interface_name>            ::=     <scoped_name>
<scoped_name>               ::=     <identifier>
                            |       "::" <identifier>
                            |       <scoped_name> "::" <identifier>
<signal_dcl>    ::= "signal" <identifier> "{" <member_list> "}"
<mediaset_dcl>      ::=     "mediaset" <identifier> "{" <member_list> "}"
<mediatype_dcl>     ::=     "mediatype" <identifier>
<medium_dcl>        ::=     "medium" <identifier> "(" <scoped_name> { "," <scoped_name> }* ")"
<value>     ::=     ( <value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl> ::=     [ "abstract" ] "valuetype" <identifier>
<value_box_dcl>     ::=     "valuetype" <identifier> <type_spec>
<value_abs_dcl>     ::=     "abstract"  "valuetype"  <identifier>  [  <value_inheritance_spec>  ]
                            "{" <export>* "}"
<value_dcl>     ::= <value_header> "{" <value_element>* "}"
<value_header>  ::=         ["custom" ] "valuetype" <identifier> [ <value_inheritance_spec> ]
<value_inheritance_spec>    ::=     [ ":" [ "truncatable" ] <value_name>
```

```
                        { "," <value_name> }* ]
                        [ "supports" <interface_name>
                        { "," <interface_name> }* ]
<value_name>    ::=  <scoped_name>
<value_element>     ::=     <export> | <state_member> | <init_dcl>
<state_member>      ::=     ( "public" | "private" ) <type_spec> <declarators> ";"
<init_dcl>    ::=  "factory" <identifier> "(" [ <init_param_decls> ] ")" ";"
<init_param_decls>  ::=     <init_param_decl> { "," <init_param_decl> }
<init_param_decl>   ::=     <init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute>      ::=     "in"
<const_dcl>     ::=         "const" <const_type> <identifier> "=" <const_exp>
<const_type>    ::=         <integer_type>
                |           <char_type>
                |           <wide_char_type>
                |           <boolean_type>
                |           <floating_pt_type>
                |           <string_type>
                |           <wide_string_type>
                |           <fixed_pt_const_type>
                |           <scoped_name>
                |           <octet_type>
<const_exp>     ::=         <or_expr>
<or_expr>       ::=         <xor_expr> | <or_expr> "|" <xor_expr>
<xor_expr>      ::=         <and_expr> | <xor_expr> "^" <and_expr>
<and_expr>      ::=         <shift_expr> | <and_expr> "&" <shift_expr>
<shift_expr>    ::=         <add_expr>
                |           <shift_expr> ">>" <add_expr>
                |           <shift_expr> "<<"<add_expr>
<add_expr>      ::=         <mult_expr>
                |           <add_expr> "+" <mult_expr>
                |           <add_expr> "-" <mult_expr>
<mult_expr>     ::=         <unary_expr>
                |           <mult_expr> "*" <unary_expr>
                |           <mult_expr> "/" <unary_expr>
                |           <mult_expr> "%" <unary_expr>
<unary_expr>    ::=         <unary_operator> <primary_expr> | <primary_expr>
<unary_operator>            ::=     "-"|"+"|"~"
<primary_expr>              ::=     <scoped_name> | <literal> | "(" <const_exp> ")"
<literal>       ::=         <integer_literal>
                |   <string_literal>
                |   <wide_string_literal>
                |   <character_literal>
```

```
                |   <wide_character_literal>
                |   <fixed_pt_literal>
                |   <floating_pt_literal>
                |   <boolean_literal>
<boolean_literal>   ::=     "TRUE"|"FALSE"
<positive_int_const> ::=    <const_exp>
<type_dcl>  ::=     "typedef" <type_declarator>
            |   <struct_type>
            |   <union_type>
            |   <enum_type>
            |   "native" <simple_declarator>
<type_declarator>   ::=     <type_spec> <declarators>
<type_spec>     ::= <simple_type_spec>
                |   <constr_type_spec>
                |   <extern_type>
<extern_type>       ::=     "extern" "type" <identifier> <string_literal>
<simple_type_spec> ::=      <base_type_spec>
                    |       <template_type_spec>
                    |       <scoped_name>
<base_type_spec>    ::=     <floating_pt_type>
                    |       <integer_type>
                    |       <char_type>
                    |       <wide_char_type>
                    |       <boolean_type>
                    |       <octet_type>
                    |       <any_type>
                    |       <object_type>
                    |       <value_base_type>
<template_type_spec>        ::=     <sequence_type>
                            |       <string_type>
                            |       <wide_string_type>
                            |       <fixed_pt_type>
<constr_type_spec> ::=      <struct_type>
                    |       <union_type>
                    |       <enum_type>
<declarators>   ::= <declarator> { "," <declarator> } *
<declarator>    ::= <simple_declarator> | <complex_declarator>
<simple_declarator> ::=     <identifier>
<complex_declarator>        ::=     <array_declarator>
<floating_pt_type>  ::=     "float"
                    |       "double"
```

```
                    |       "long" "double"
<integer_type>  ::= <signed_int> | <unsigned_int>
<signed_int>    ::= <signed_short_int>
                  | <signed_long_int>
                  | <signed_longlong_int>
<signed_short_int>  ::=     "short"
<signed_long_int>   ::=     "long"
<signed_longlong_int>       ::=     "long" "long"
<unsigned_int>      ::=     <unsigned_short_int>
                    |       <unsigned_long_int>
                    |       <unsigned_longlong_int>
<unsigned_short_int>        ::=     "unsigned" "short"
<unsigned_long_int>         ::=     "unsigned" "long"
<unsigned_longlong_int>             ::=     "unsigned" "long" "long"
<char_type>     ::= "char"
<wide_char_type>    ::=     "wchar"
<boolean_type>      ::=     "boolean"
<octet_type>    ::= "octet"
<any_type>      ::= "any"
<object_type>   ::= "Object"
<struct_type>   ::= "struct" <identifier> "{" <member_list> "}"
<member_list>   ::= <member>+
<member>    ::= <type_spec> <declarators> ";"
<union_type>    ::= "union" <identifier> "switch" "(" <switch_type_spec> ")" "{" <switch_body> "}"
<switch_type_spec>          ::=     <integer_type>
                            |       <char_type>
                            |       <boolean_type>
                            |       <enum_type>
                            |       <scoped_name>
<switch_body>               ::=     <case>+
<case>          ::=         <case_label>+ <element_spec> ";"
<case_label>    ::=         "case" <const_exp> ":"|"default" ":"
<element_spec> ::=          <type_spec> <declarator>
<enum_type>     ::=         "enum" <identifier> "{" <enumerator> { "," <enumerator> } * "}"
<enumerator>    ::=         <identifier>
<sequence_type>             ::=     "sequence" "<" <simple_type_spec> ","
                            <positive_int_const> ">" | "sequence" "<" <simple_type_spec> ">"
<string_type>   ::= "string" "<" <positive_int_const> ">" | "string"
<wide_string_type>  ::=     "wstring" "<" <positive_int_const> ">" | "wstring"
<array_declarator>  ::=     <identifier> <fixed_array_size>+
<fixed_array_size>  ::=     "[" <positive_int_const> "]"
```

```
<attr_dcl>    ::=       [ "readonly" ] "attribute"
                        <param_type_spec> <simple_declarator> { "," <simple_declarator> }*
<except_dcl>     ::=  "exception" <identifier> "{" <member>* "}"
<op_dcl>     ::=  [ <op_attribute> ] <op_type_spec>
                  <identifier> <parameter_dcls> [ <raises_expr> ] [ <context_expr> ]
<op_attribute>          ::=       "oneway"
<op_type_spec>          ::=       <param_type_spec> |"void"
<parameter_dcls>        ::=       "(" <param_dcl> { "," <param_dcl> } * ")"|"(" ")"
<param_dcl>     ::=  <param_attribute> <param_type_spec> <simple_declarator>
<param_attribute>       ::=       "in"|"out"|"inout"
<raises_expr>    ::=  "raises" "(" <scoped_name>{ "," <scoped_name> } * ")"
<context_expr>  ::=               "context" "(" <string_literal>{ "," <string_literal> } * ")"
<param_type_spec> ::=     <base_type_spec>
                        |         <string_type>
                        |         <wide_string_type>
                        |         <scoped_name>
<fixed_pt_type>         ::=       "fixed" "<" <positive_int_const> ","<positive_int_const> ">"
<fixed_pt_const_type>           ::=     "fixed"
<value_base_type>  ::=    "ValueBase"
<assembly_dcl> ::=  <assembly_header> "{"  <assembly_body>  "}"
<assembly_header> ::=     "assembly"  <identifier>
<assembly_body>    ::=    <assembly_stmt>*
<assembly_stmt>    ::=    <instance_set_dcl> ";"
                        |         <connect_dcl> ";"
                        |         <constraint_dcl> ";"
                        |         <property_list>
<instance_set_dcl>  ::=   <identifier> [ "(" <integer_literal> ")" ] ":" <cotype_identifier>
<cotype_identifier>  ::=  <scoped_name>
<connect_dcl>       ::=   "connect"  [ <identifier> ]  "{"  <connection_list>  "}"
<connection_list>    ::=  { <connection> ";" } +
<connection>        ::=   <instance_set_identifier>  "."  <port_identifier>
                          "=" <instance_set_identifier>  "."  <port_identifier>
<instance_set_identifier>       ::=     <scoped_name>
<port_identifier>       ::=       <scoped_name>
<softwarecomponent_dcl>     ::=     <softwarecomponent_header>
                                    "{" <softwarecomponent_body> "}"
<softwarecomponent_header>          ::=         "softwarecomponent"  <identifier>
                                                "realizes"  <cotype_identifier_list>
<cotype_identifier_list>        ::=     <cotype_identifier> { ","  <cotype_identifier> }*
<softwarecomponent_body>  ::=     <softwarecomponent_stmt>*
<softwarecomponent_stmt>   ::=    "dependent"  "{"  <softwarecomponent_list>  "}" ";"
```

```
                              |       "requires"  "{"  <property_list>  "}"  ";"
<softwarecomponent_list>      ::=     <softwarecomponent_identifier>
                                      {  ","  <softwarecomponent_identifier>  }*
<softwarecomponent_identifier>        ::=         <scoped_name>
<environment_dcl>  ::=        <environment_header> "{"  <environment_body>  "}"
<environment_header>          ::=     "environment"  <identifier>
<environment_body> ::=        <environment_stmt>+
<environment_stmt>  ::=       <node_dcl>  ";"
                        |     <link_dcl>  ";"
<node_dcl> ::=        "node"  <identifier>  "{"  <property_list>  "}"
<property_list>       ::=     { <property_dcl> ";" }*
<property_dcl>        ::=     <property_name>  "="  <property_value>
<property_name>       ::=     <identifier>
<property_value>      ::=     <simple_property_value>
                      |       <structured_property_value>
                      |       <sequence_property_value>
<simple_property_value>       ::=     <string_literal>
                      |       <integer_literal>
                      |       <boolean_literal>
<structured_property_value>   ::=     "{"  <property_assign>*  "}"
<sequence_property_value>     ::=     "["  <property_value>*  "]"
<constraint_dcl>      ::=     "constraint" <identifier> "{" <constraint_body> "}"
<constraint_body>     ::=     "language" "=" <string_literal> "body" "=" <string_literal> ";"
<property_assign>     ::=     <property_name> "="  <property_value> ";"
<link_dcl>        ::=  <link_header>  "{"  <link_body>  "}"
<link_header>     ::=  "link"  <identifier>
<link_body>       ::=  "node" <node_list>  ";" <property_list> ";"
<node_list>       ::=  <node_identifier>  {  ","  <node_identifier>  }*
<node_identifier>     ::=     <scoped_name>
<installation_map_dcl>        ::=     <installation_map_header>  "{" <installation_map_body> "}"
<installation_map_header>     ::=     "installation" <identifier>
                                      "uses" "environment" <environment_identifier>
<installation_map_body>       ::=     <install_stmt>*
<install_stmt>                ::=     <softwarecomponent_identifier> "->" <node_identifier> ";"
<environment_identifier>      ::=     <scoped_name>
<instantiation_map_dcl>       ::=     <instantiation_map_header> "{"  <instantiation_map_body>  "}"
<instantiation_map_header>    ::=     "instantiation" <identifier>
                                      <instantiation_map_header_env>
                                      <instantiation_map_header_ass>
<instantiation_map_header_env>        ::=         "uses"  "environment"  <environment_identifier>
<instantiation_map_header_ass>        ::=         "uses" "assembly" <assembly_identifier>
<assembly_identifier>         ::=     <scoped_name>
```

```
<instantiation_map_body>     ::=    <assign_instance_stmt>*
<assign_instance_stmt>       ::=    <instance_set_identifier_list> "->" <node_identifier> ";"
<instance_set_identifier_list> ::=  <instance_set_identifier> { "," <instance_set_identifier> }*
<deployment_action> ::=      "deploy" "{" <deployment_body> "}" ";"
<deployment_body>  ::=       "install" "{" <install_list> "}" ";"
                             "instantiate" "{" <instantiation_list> "}" ";"
<install_list>     ::= <install_member>*
<install_member>   ::=       <installation_map_identifier>  ";"
                         |   <qualified_install_stmt>
<qualified_install_stmt>     ::=    <softwarecomponent_identifier>  "->"
                                    <environment_identifier>  "."  <node_identifier>  ";"
<installation_map_identifier> ::=   <scoped_name>
<instantiation_list>  ::=    <instantiation_member>*
<instantiation_member>       ::=    <instantiation_map_identifier>  ";"
                             |      <qualified_assign_instance_stmt> ";"
<instantiation_map_identifier>       ::=       <identifier>
<qualified_assign_instance_stmt>  ::=          <assembly_identifier>  "."
                                               <instance_set_identifier>  "->"
                                               <environment_identifier>  "."  <node_identifier>
```

Annex B

Metamodel to syntax mapping

B.1 Introduction

This annex describes the relation between the eODL **metamodel** and the concrete textual syntax of eODL as defined in Annex A. The description is restricted to those **metamodel** concepts being extensions of the CORBA **metamodel**. The relation between OMG IDL 2.4.2 textual syntax and OMG CORBA **metamodel** are well defined by OMG and therefore not repeated here.

The **metamodel** is as specified in clause 5, and uses the graphical notation therein. The corresponding alphanumeric syntax is provided in the box beneath the graph, followed by a textual explanation.

B.2 Signal and Signal Parameter

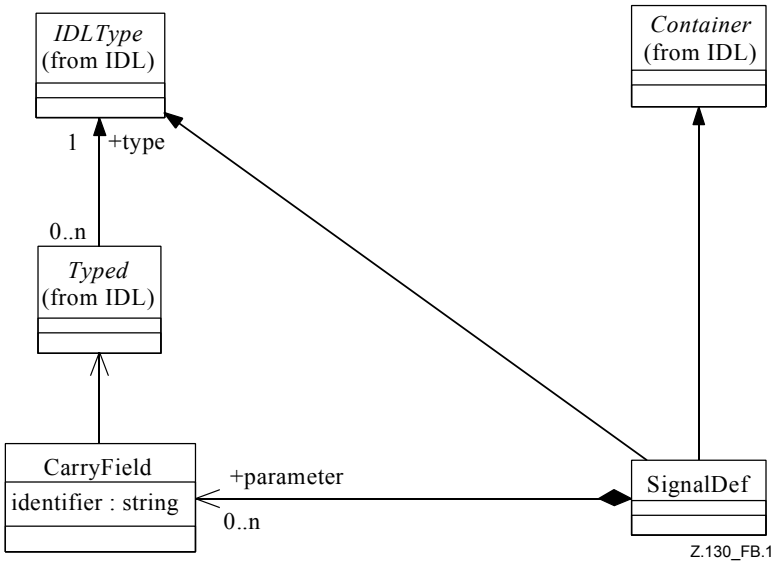


Figure B.1/Z.130 – Signal and Signal Parameter

The **(1)**, **(2)** and **(3)** in the concrete syntax are mapped to *SignalDef*/*CarryField* elements in the model (see Figure B.1).

(1) **<signal_dcl>** **::=** **"signal" <identifier> "{" <member_list> "}"**

(2) **<member_list>** **::=** **<member>+**

(3) **<member>** **::=** **<type_spec> <declarators> ";"**

The **<identifier>** from production **(1)** is the name of the *SignalDef* element in the model. Within the **<member_list> (2)** all *CarryField* elements are listed, which take part in the **parameter** relation of that *SignalDef*. The **<member> (3)** productions from concrete syntax are reflected as *CarryField* elements in the model. **<type_spec>** here are the types in the model, which are bounded through the Typed concept from IDL. For each declarator in **<declarators>** a *CarryField* element in the model exists.

B.3 Medium Type, Medium, Media Set

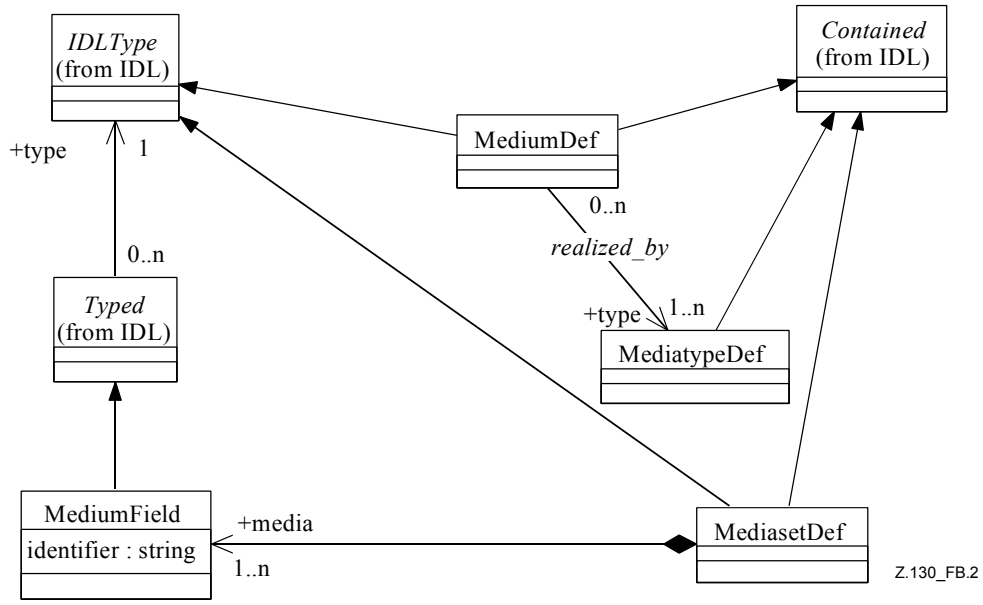


Figure B.2/Z.130 – Medium, Media type, Mediaset

The productions **(4)**, **(5)** and **(6)** in the concrete syntax are mapped to *MediasetDef*, *MediatypeDef* and *MediumDef* elements in the model (see Figure B.2).

(4) **<mediaset_dcl> ::=** **"mediaset" <identifier> "{" <member_list> "}"**

(5) **<mediatype_dcl>::=** **"mediatype" <identifier>**

(6) **<medium_dcl> ::=** **"medium" <identifier>**
"(" <scoped_name> { "," <scoped_name> }* ")"

The **<mediatype_dcl> (5)** is represented as *MediatypeDef* element in the model; **<identifier>** is the name for the Named concept of this element. With **<medium_dcl>** the concrete syntax express *MediumDef* elements; here **<identifier>** again is the name for the Named concept. The **<scoped_name>** listed in production **(6)** have to refer always to *MediatypeDef* and are represented in the model as *realized_by* relation to the *MediumDef* element. According to production **(4) <mediaset_dcl>** is represented as *MediasetDef* element in the model with **<identifier>** as name. Within the **<member_list>** in **(4)** all *MediumField* elements are listed, which take part in the media relation of that *MediasetDef*. For the **<member>** in **<member_list> (4)** the **<type_spec>** has to refer to *MediatypeDef* and all declarators in **<declarators>** should be simple.

B.4 Consume and Produce

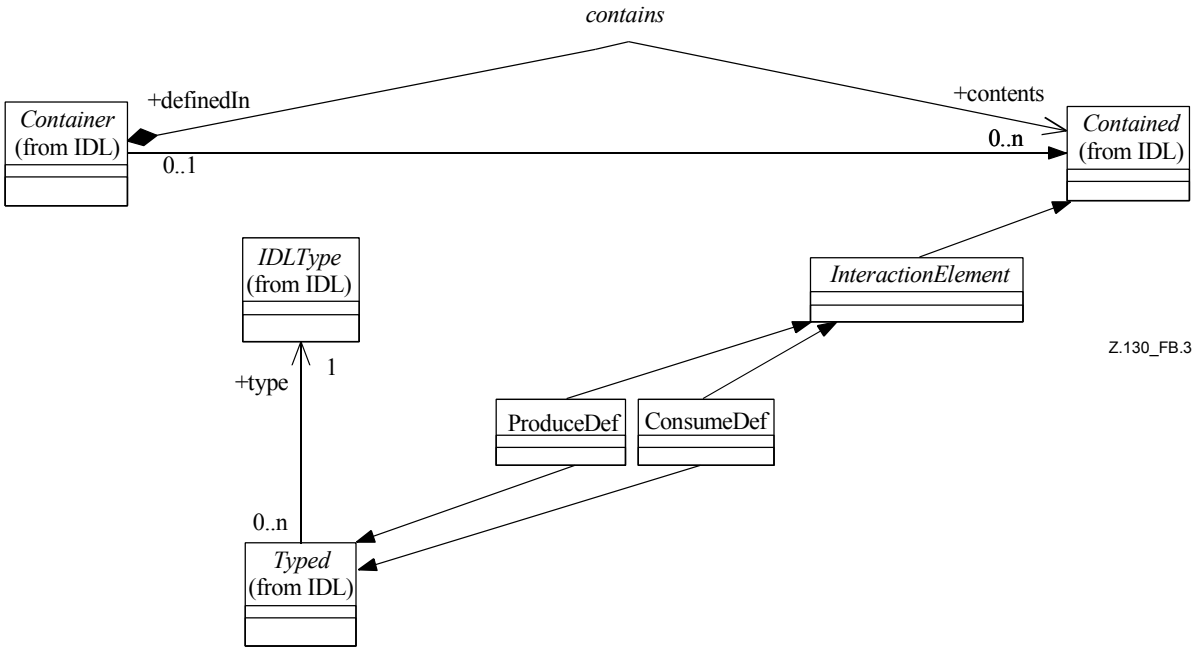


Figure B.3/Z.130 – Consume and Produce

The productions **(7)** and **(8)** in the concrete syntax are mapped to *ProduceDef/ConsumeDef* elements in the model (see Figure B.3).

(7) <produce_dcl> ::= "produce" <scoped_name> <identifier>

(8) <consume_dcl> ::= "consume" <scoped_name> <identifier>

The **<scoped_name>** in both declarations refers to a *SignalDef*. This relation is reflected in the model as the *Typed/IDLType* relation where *ProduceDef/ConsumeDef* are involved via inheritance and the *IDLType* is a *SignalDef* according to the **<signal_dcl>**. *ProduceDef/ConsumeDef* are both Named concepts from the **metamodel** and the **<identifier>** in **(7)**/**(8)** is mapped to the name attribute of the model elements.

B.5 Sink and Source

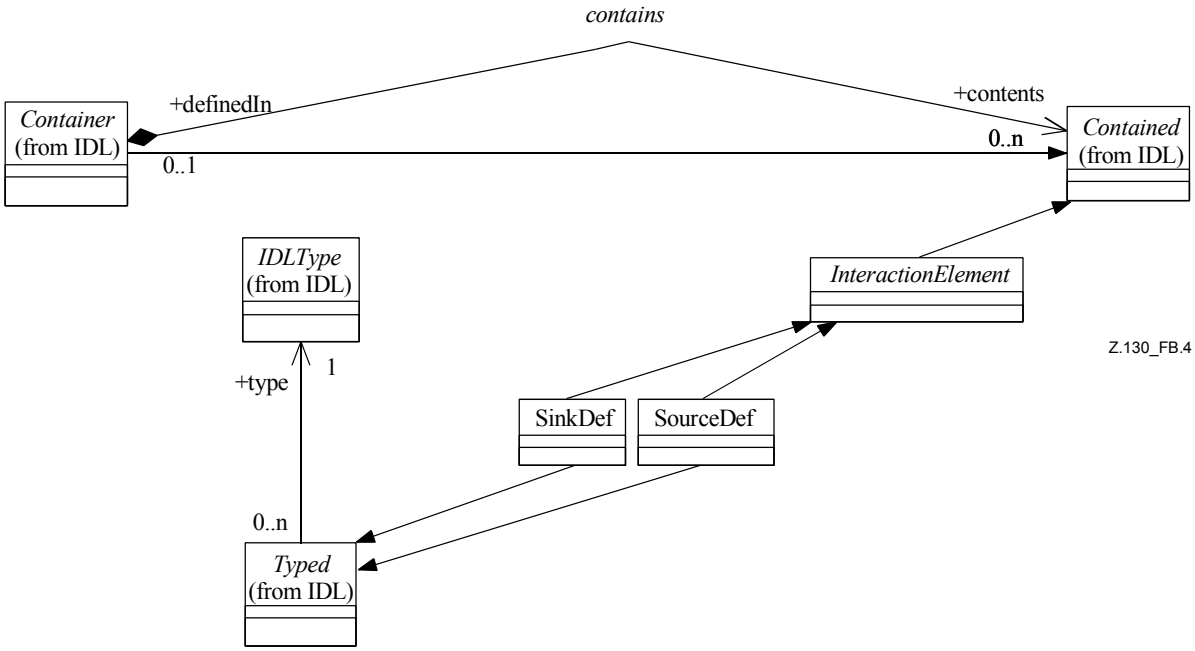


Figure B.4/Z.130 – Sink and Source

The productions **(10)** and **(9)** in the concrete syntax are mapped to *SinkDef/SourceDef* elements in the model (see Figure B.4).

(9) **<source_dcl>** **::=** **"source" <scoped_name> <identifier>**

(10) **<sink_dcl>** **::=** **"sink" <scoped_name> <identifier>**

The **<scoped_name>** in both declarations refers to a *MediasetDef*. This relation is reflected in the model as the *Typed/IDLType* relation where *SinkDef/SourceDef* are involved via inheritance and the *IDLType* is a *MediasetDef* according to the **<mediaset_dcl>**. *SinkDef/SourceDef* are both Named concepts from the **metamodel** and the **<identifier>** in **(10)/(9)** is mapped to the name attribute of the model elements.

B.6 Interface Type

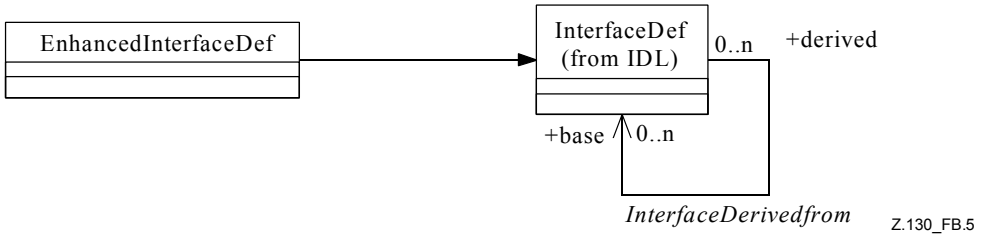


Figure B.5/Z.130 – Interface Type

(11) **<interface_dcl>** **::=** **<interface_header> "{" <interface_body> "}"**

(12) **<interface_body>** **::=** **<export> ***

(13) **<export>** **::=** **<type_dcl> ";"**
| **<const_dcl> ";"**
| **<except_dcl> ";"**
| **<attr_dcl> ";"**
| **<op_dcl> ";"**
| **<produce_dcl> ";"**
| **<consume_dcl> ";"**
| **<source_dcl> ";"**
| **<sink_dcl> ";"**

The **<interface_dcl> (11)** production in concrete syntax is mapped to an *EnhancedInterfaceDef* element in the model. **<interface_body> (12)** is handled in the same way as in IDL. In comparison to *InterfaceDef* **<export> (13)** also allows **<produce_dcl>**, **<consume_dcl>**, **<source_dcl>** and **<sink_dcl>** as contained elements. Mapping for this kind of declarations is defined above. If the **<interface_body>** does not contain this kind of new elements, the **<interface_dcl>** is mapped to an ordinary *InterfaceDef* (see Figure B.5).

B.7 CO Types, Supports and Requires

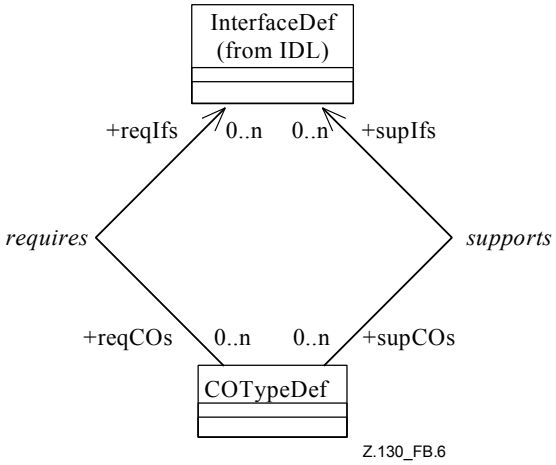


Figure B.6/Z.130 – CO Types, Supports and Requires

The productions **(14)**, **(15)**, and **(16)** in the concrete syntax are mapped to *COTypeDef* elements in the model (see Figure B.6).

(14) **<object_template>** **::=** **<object_template_header> "{" <object_template_export> "}"**
(15) **<object_template_header>** **::=** **"CO" <identifier> [<object_inheritance_spec>]**
(16) **<object_inheritance_spec>** **::=** **":" <scoped_name> { "," <scoped_name> }***
(17) **<object_template_export>** **::=** **<object_export>***
(18) **<object_export>** **::=** **<export>**
| **<reqrd_interf_templates> ";"**
| **<suptd_interf_templates> ";"**
| **<use_dcl> ";"**
| **<provide_dcl> ";"**
| **<implements_dcl> ";"**
| **<state_def_dcl> ";"**

| **<constraint_dcl> ";"**

| **<property_list>**

(19) **<reqrd_interf_templates>** **::=** **"requires" <scoped_name> { "," <scoped_name> }***

(20) **<supd_interf_templates>** **::=** **"supports" <scoped_name> { "," <scoped_name> }***

Productions **(14)**, **(15)** and **(16)** map to a *COTypeDef* element in the model, where **<identifier>** from production **(15)** is the name for the Named concept of *COTypeDef*. As specialization of *InterfaceDef* in production **(16)** all *COTypeDef* are listed which are in inheritance relation to the current *COTypeDef*. Productions **(17)** and **(18)** express the contained concept for this *COTypeDef*. The use of **<export>** in **(18)** is handled in the same way as for *InterfaceDef* in IDL. In addition **<reqrd_interf_templates>** and **<supd_interf_templates>** are allowed contained elements for *COTypeDef*. The **<scoped_name>** in productions **(19)** and **(20)** has to refer to *InterfaceDef* or *EnhancedInterfaceDef* elements in the model. These interface elements are in **requires** or **supports** relation to the containing *COTypeDef* element.

B.8 Provided and Used Port

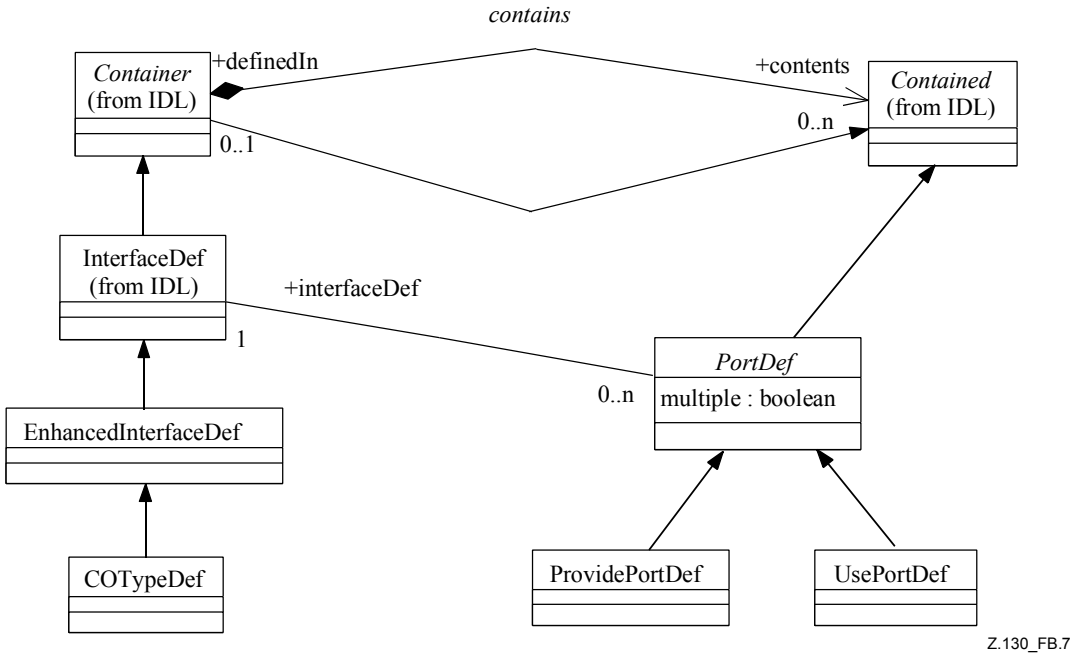


Figure B.7/Z.130 – Provided and Used Port

The productions **(21)**/**(22)** in the concrete syntax are mapped to *UsePortDef/ProvidePortDef* elements in the model (see Figure B.7).

(21) **<use_dcl>** **::=** **"use" ["multiple"] <scoped_name> <identifier>**

(22) **<provide_dcl>** **::=** **"provide" ["multiple"] <scoped_name> <identifier>**

Used and **provided ports** are expressed with productions **(21)** and **(22)**. They result in *UsedPortDef* and *ProvidePortDef* elements in the model. The **<scoped_name>** in both productions has to refer to *InterfaceDef* or *EnhancedInterfaceDef* elements in the model. If "multiple" is used in the concrete syntax, the multiple Boolean field in the current *PortDef* is true.

B.9 Artefact and Instantiation Pattern

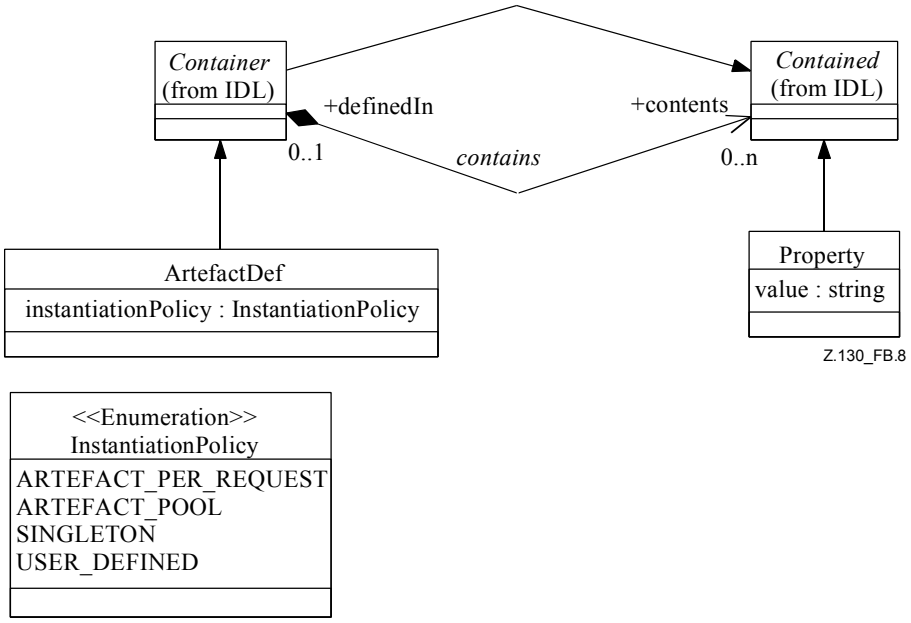


Figure B.8/Z.130 – Artefact and Instantiation Pattern

(23)	**<artefact>**	**::=**	**<artefact_dcl>**
		|	**<artefact_forward_dcl>**
(24)	**<artefact_forward_dcl>**	**::=**	**"artefact" <identifier>**
(25)	**<artefact_dcl>**	**::=**	**<artefact_header> "{" <artefact_body> "}"**
(26)	**<artefact_header>**	**::=**	**"artefact" <identifier> [<artefact_inheritance_spec>]**
(27)	**<artefact_inheritance_spec>**	**::=**	**":" <scoped_name> { "," <scoped_name> }***
(28)	**<artefact_body>**	**::=**	**<impl_elem_dcl>***

Productions **(23)** and **(24)** are used to follow the syntax of IDL, where the **<identifier>** used in **(24)** has a following *ArtefactDef* with this name. Productions **(25)**, **(26)** and **(27)** are in relation to an *ArtefactDef* element in model, where **<identifier>** from **(26)** is the name for the Named concept. All listed **<scoped_name>** in **(27)** have to refer to *ArtefactDef* elements in the model, which are in inheritance relation to the current *ArtefactDef*. As **(28)** shows, only *ImplementationElementDef* is allowed to be contained in *ArtefactDef* (see Figure B.8).

B.10 Implements Relation

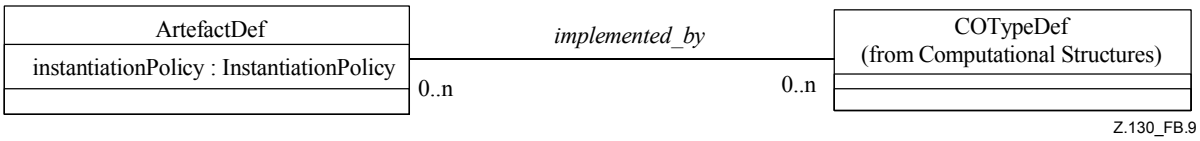


Figure B.9/Z.130 – Implements Relation

(29)	**<implements_dcl>**	**::=**	**"implemented" "by" <artefact_with_policy>**
			{ "," <artefact_with_policy> }*
(30)	**<artefact_with_policy>**	**::=**	**<scoped_name> ["with" <instantiation_policy_dcl>]**

(31) **<instantiation_policy_dcl>** **::=** **"ArtefactPool"**

| **"ArtefactPerRequest"**

| **"Singleton"**

| **"UserDefined"**

Productions **(29)** and **(30)** are in relation to an *ArtefactDef* element in the model (see Figure B.9). This expresses the *implemented_by* relation in the model. The **<scoped_name>** in **(30)** has to refer to an *ArtefactDef* element only. With production **(31)** in the concrete syntax, the *instantiationPolicy* field of the containing *ArtefactDef* element is expressed. The keywords directly relate to the enumeration values for the field.

B.11 Implementation Element

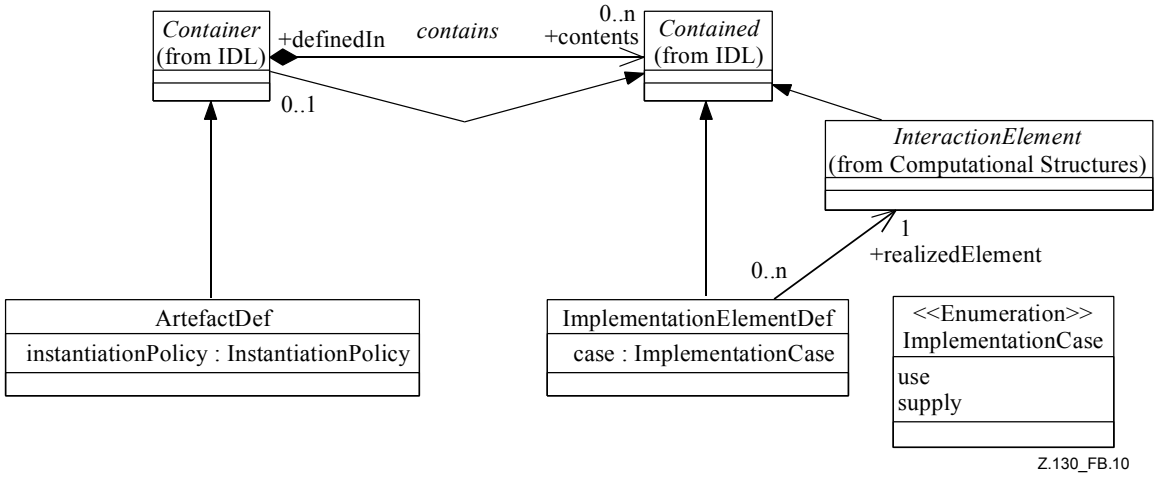


Figure B.10/Z.130 – Implementation Element

(32) **<impl_elem_dcl>** **::=** **<identifier> "implements" <impl_case_dcl> <scoped_name> ";"**

(33) **<impl_case_dcl>** **::=** **"supply" | "use"**

Productions **(32)** and **(33)** are in relation to an *ImplementationElementDef* element in the model (see Figure B.10). The **<identifier>** in **(32)** is the name for the Named concept. *ImplementationElementDef* can only be containted in *ArtefactDef*. With production **(33)** in the concrete syntax, the *case* field of the containing *ImplementationElementDef* element is expressed. The keywords directly relate to the enumeration values for the field.

B.12 Software Component

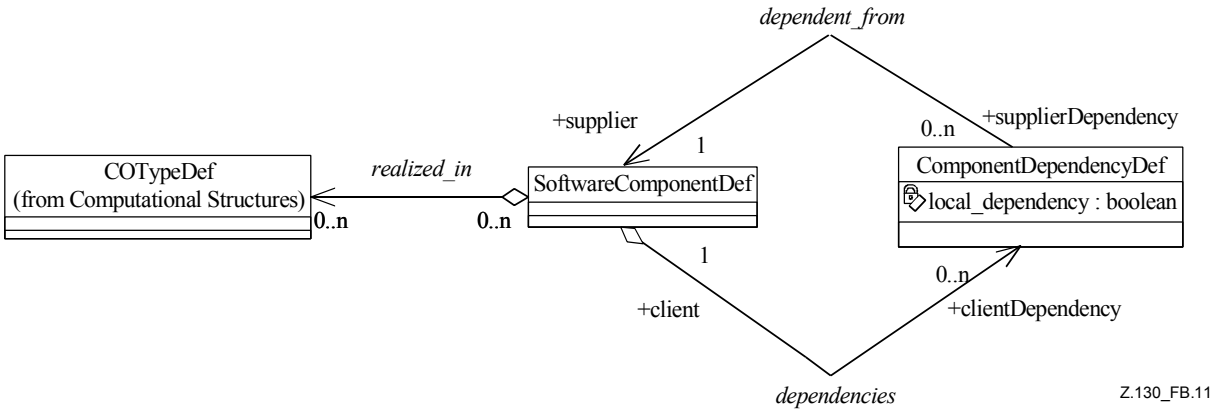


Figure B.11/Z.130 – Software Component

The production **(34)** in the concrete syntax is mapped to *SoftwareComponentDef* element in the model (see Figure B.11).

(34) **<softwarecomponent_dcl>** **::=** **<softwarecomponent_header>**
"{" <softwarecomponent_body> "}"

(35) **<softwarecomponent_header>** **::=** **"softwarecomponent" <identifier> "realizes"**
<cotype_identifier_list>

(36) **<cotype_identifier_list>** **::=** **<cotype_identifier> { "," <cotype_identifier> }***

(37) **<softwarecomponent_body>** **::=** **<softwarecomponent_stmt>***

(38) **<softwarecomponent_stmt>** **::=** **"dependent" "{" <softwarecomponent_list> "}" ";"**
| **"requires" "{" <property_list> "}" ";**

(39) **<softwarecomponent_list>** **::=** **<softwarecomponent_identifier>**
{ "," <softwarecomponent_identifier> }*

Productions **(34)**, **(35)** and **(36)** are in relation to a *SoftwareComponentDef* element in the model, where **<identifier>** from **(35)** is the name for the Named concept. All listed **<scoped_name>** in **(36)** have to refer to *COTypeDef* elements in the model, which are in *realized_in* relation to the current *SoftwareComponentDef*. Productions **(37)**, **(38)** and **(39)** are in relation to a *SoftwareComponentDef* element in the model. All listed **<scoped_name>** in **(36)** have to refer to *SoftwareComponentDef* elements in the model.

The **<softwarecomponent_identifier>** is a **<scoped_name>**, which refers only to *SoftwareDependencyDef* model element.

B.13 Assembly and Initial Configuration

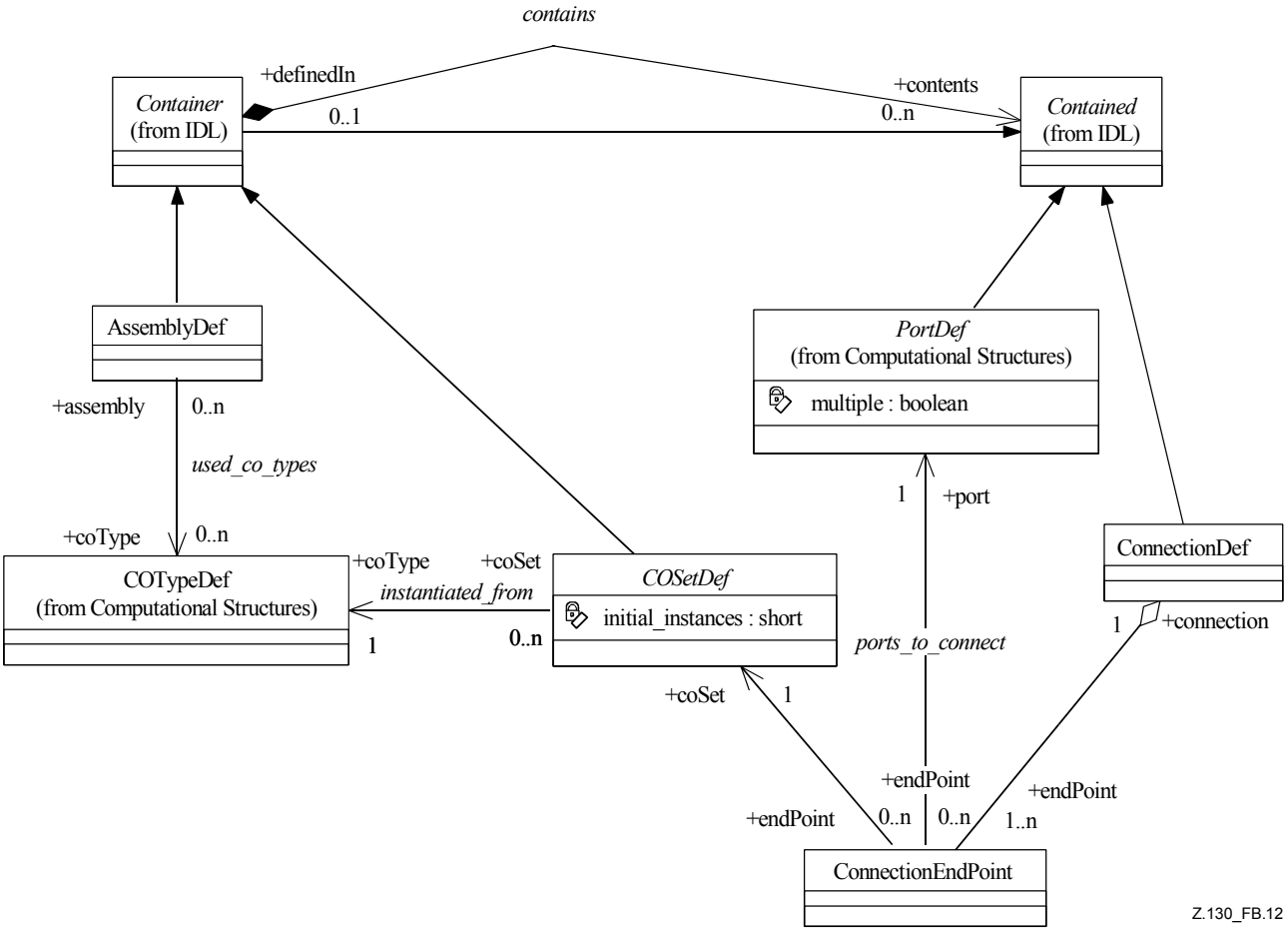


Figure B.12/Z.130 – Assembly and Initial Configuration

(40)	**<assembly_dcl>**	**::=**	**<assembly_header> "{" <assembly_body> "}"**
(41)	**<assembly_header>**	**::=**	**"assembly" <identifier>**
(42)	**<assembly_body>**	**::=**	**<assembly_stmt>***
(43)	**<assembly_stmt>**	**::=**	**<instance_set_dcl>";"**
		|	**<connect_dcl> ";"**
		|	**<constraint_dcl> ";"**
		|	**<property_list>**
(44)	**<instance_set_dcl>**	**::=**	**<identifier> ["(" <integer_literal> ")"] ":" <cotype_identifier>**
(45)	**<connect_dcl>**	**::=**	**"connect" [<identifier>] "{" <connection_list> "}"**
(46)	**<connection_list>**	**::=**	**{ <connection> ";" } +**
(47)	**<connection>**	**::=**	**<instance_set_identifier> "." <port_identifier>**
			"=" <instance_set_identifier> "." <port_identifier>

Productions **(40)** and **(41)** are in relation to an *AssemblyDef* element in the model (see Figure B.12), where **<identifier>** from **(41)** is the name for the Named concept. All listed **<scoped_name>** in **(44)** have to refer to *COTypeDef* elements in the model, which are in *realized_in* relation to the current *SoftwareComponentDef*. The **<cotype_identifier>**, **<instance_set_identifier>** and **<port_identifier>** in **(45)**, **(46)** and **(47)** are **<scoped_name>**s, which refer only to *COTypeDef*, *InstanceSetDef* and *PortDef* model elements.

B.14 Constraints and Properties

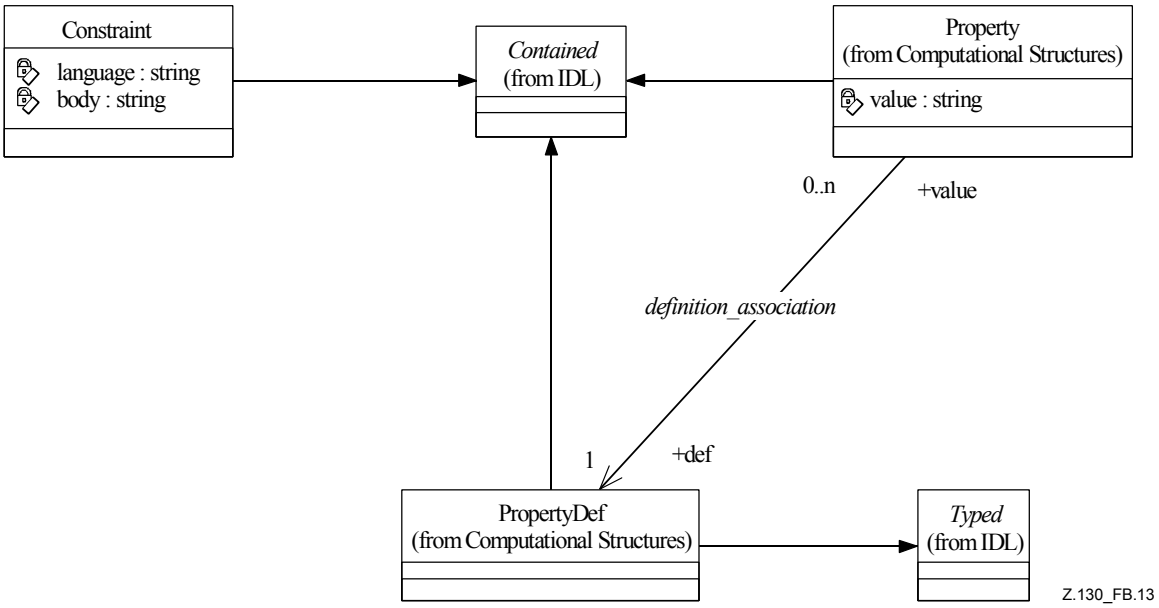


Figure B.13/Z.130 – Constraints and Properties

(48) **<property_dcl>** **::=** **"property" <property_name> "=" <property_value>**

(49) **<property_name>** **::=** **<identifier>**

(50) **<property_value>** **::=** **<simple_property_value>**

| **<structured_property_value>**

| **<sequence_property_value>**

(51) **<simple_property_value>** **::=** **<string_literal>**

| **<integer_literal>**

| **<boolean_literal>**

(52) **<structured_property_value>::=** **"{" <property_assign>* "}"**

(53) **<sequence_property_value>::=** **"[" <property_value>* "]"**

(54) **<property_assign>** **::=** **<property_name> "=" <property_value> ";"**

(55) **<constraint_dcl>** **::=** **"constraint" <identifier> "{" <constraint_body> "}"**

(56) **<constraint_body>** **::=** **"language" "=" <string_literal> "body" "=" <string_literal> ";"**

Productions **(48)** and **(49)** are in relation to a *PropertyDef* element in the model (see Figure B.13), where the **<identifier>** from **(49)** is the name for the Named concept. Production **(54)** maps to a *Property* element in the model. Productions **(50)**, **(51)**, **(52)** and **(53)** in the concrete textual syntax are used to notate values for the value *field*.

Production **(55)** is in relation to a *Constraint* element in the model, where the **<identifier>** from **(55)** is the name for the Named concept. Production **(56)** provides the values for the fields *language* and *body* in the *Constraint* element.

B.15 Target Environment, Node and NodeLink

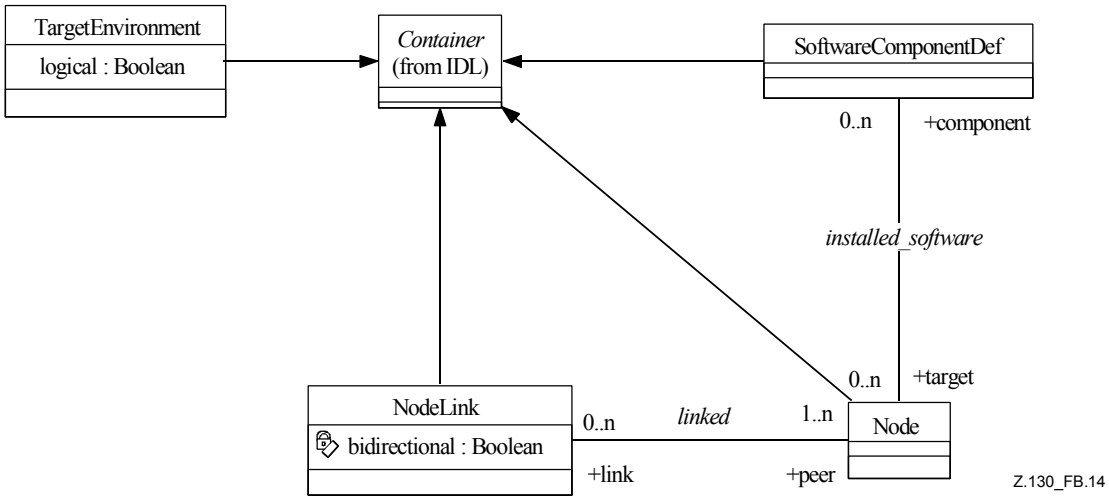


Figure B.14/Z.130 – Target Environment, Node and NodeLink

(57)	**<environment_dcl>**	**::=**	**<environment_header> "{" <environment_body> "}"**
(58)	**<environment_header>**	**::=**	**"environment" <identifier>**
(59)	**<environment_body>**	**::=**	**<environment_stmt>+**
(60)	**<environment_stmt>**	**::=**	**<node_dcl> ";"**
		|	**<link_dcl> ";"**
(61)	**<node_dcl>**	**::=**	**"node" <identifier> "{" <property_list> "}"**
(62)	**<link_dcl>**	**::=**	**<link_header> "{" <link_body> "}"**
(63)	**<link_header>**	**::=**	**"link" <identifier>**
(64)	**<link_body>**	**::=**	**"node" <node_list> ";" <property_list>**
(65)	**<node_list>**	**::=**	**<node_identifier> { "," <node_identifier> }+**

Productions **(57)**, **(58)**, **(59)** and **(60)** are in relation to a *TargetEnvironment* element in the model (see Figure B.14), where the **<identifier>** in production **(58)** is the name for the Named concept. Production **(61)** maps to a *Node* element in the model, where the **<identifier>** in production **(61)** is the name for the Named concept. Productions **(62)**, **(63)** and **(64)** are in relation to a *NodeLink* element in the model, where the **<identifier>** in production **(63)** is the name for the Named concept. The **<node_identifier>** in production **(65)** is a **<scoped_name>**, which has to refer to a *Node* model element.

B.16 InstallationMap

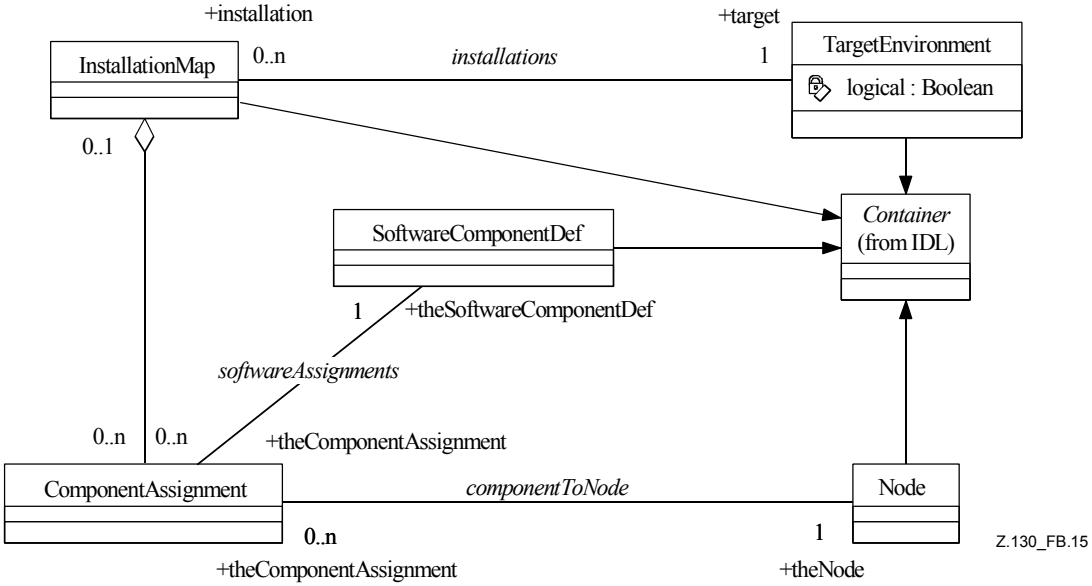


Figure B.15/Z.130 – InstallationMap

(66) **<installation_map_dcl>** **::=** **<installation_map_header>**
"{" <installation_map_body> "}"

(67) **<installation_map_header>** **::=** **"installation" <identifier> "uses" "environment"**
<environment_identifier>

(68) **<installation_map_body>** **::=** **<install_stmt>***

(69) **<install_stmt>** **::=** **<softwarecomponent_identifier> "->" <node_identifier> ";"**

Productions **(66)**, **(67)** and **(68)** are in relation to an *InstallationMap* element in the model (see Figure B.15), where the **<identifier>** in production **(67)** is the name for the Named concept. The **<environment_identifier>** in production **(67)** is a **<scoped_name>**, which refers only to the *TargetEnvironment* model element. The **<softwarecomponent_identifier>** and **<node_identifier>** in production **(69)** are **<scoped_name>**s, which refer only to *SoftwareComponentDef* and *Node* elements in the model.

B.17 InstantiationMap

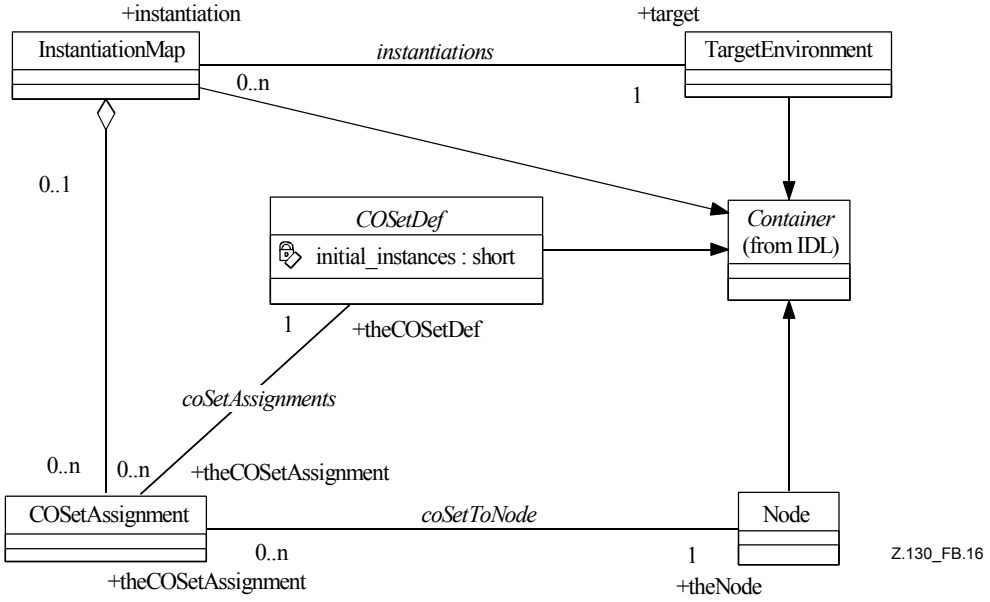


Figure B.16/Z.130 – InstantiationMap

(70)	**<instantiation_map_dcl>**	**::=**	**<instantiation_map_header>**
			"{" <instantiation_map_body> "}"
(71)	**<instantiation_map_header>**	**::=**	**"instantiation" <identifier>**
			<instantiation_map_header_env>
			<instantiation_map_header_ass>
(72)	**<instantiation_map_header_env>**	**::=**	**"uses" "environment" <environment_identifier>**
(73)	**<instantiation_map_header_ass>**	**::=**	**"uses" "assembly" <assembly_identifier>**
(74)	**<instantiation_map_body>**	**::=**	**<assign_instance_stmt>***
(75)	**<assign_instance_stmt>**	**::=**	**<instance_set_identifier_list> "->"**
			<node_identifier> ";"
(76)	**<instance_set_identifier_list>**	**::=**	**<instance_set_identifier>**
			{ "," <instance_set_identifier> }*

Productions **(70)**, **(71)**, **(72)**, **(73)** and **(74)** are in relation to an *InstantiationMap* element in the model (see Figure B.16), where the **<identifier>** in production **(71)** is the name for the Named concept. The **<environment_identifier>** in production **(72)** and **<assembly_identifier>** in production **(73)** are **<scoped_name>**s, which refer only to *TargetEnvironment* and *AssemblyDef* elements in the model. The **<node_identifier>** in production **(75)** is a **<scoped_name>**, which refers only to *Node* model element, which is contained in the *TargetEnvironment* qualified by production **(72)**. The **<instance_set_identifier>**s in production **(76)** are **<scoped_name>**s, which refer only to *COSetDef* model elements, which are contained in the *AssemblyDef* qualified by production **(73)**.

B.18 Deployment Plan

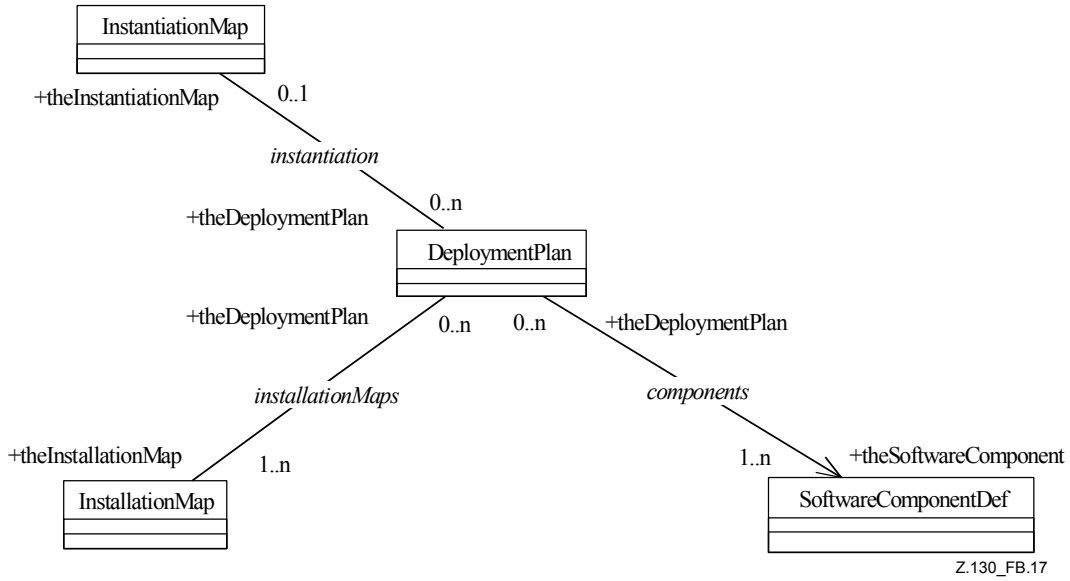


Figure B.17/Z.130 – Deployment Plan

(77) **<deployment_action>** **::=** **"deploy" "{" <deployment_body> "}" ";"**

(78) **<deployment_body>** **::=** **"install" "{" <install_list> "}" ";" <instantiation_action>+**

(79) **<install_list>** **::=** **<install_member>***

(80) **<install_member>** **::=** **<installation_map_identifier> ";"**

(81) **<instantiation_action>** **::=** **"instantiate" <instantiation_map_identifier> ";"**

The production **(77) <deployment_action>** in concrete syntax maps to the *DeploymentPlan* element in the model (see Figure B.17). The lists in the body of the **<deployment_action>**, which are built by productions **(78)**, **(79)**, **(80)** and **(81)**, express the *InstallationMaps* and instantiation relations in the model. The **<instantiation_map_identifier>** and **<installation_map_identifier>** are **<scoped_name>**s, which refer to *InstallationMap* and *InstantiationMap* model elements. Each of the listed identifiers corresponds to a relation in the model.

B.19 Extern type

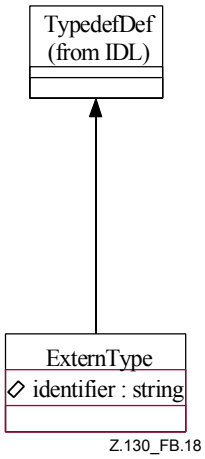


Figure B.18/Z.130 – Extern type

(82) <extern_type> ::= "extern" "type" <identifier> <string_literal>";"

The production **(82) <extern_type>** in concrete syntax maps to the *ExternType* element in the model (see Figure B.18). The value of the string literal is directly mapped onto the attribute identifier of the model element. The **<identifier>** in production **(82)** is mapped onto the name attribute of the concept *Contained*.

Annex C

Mapping to SDL-2000

C.1 Introduction

The recommended ITU Extended Object Definition Language (ITU-eODL) provides the ability to describe component-oriented distributed systems. In this annex, an ITU-eODL to SDL-2000 mapping is introduced. This mapping allows to generate SDL-2000 code based upon a description given in ITU-eODL automatically. The textual phrase representation (SDL/PR) of SDL-2000 [8] is used.

The mapping from ITU-eODL to SDL-2000 allows users to generate a SDL-2000 skeleton based on a given eODL model automatically. The mapping supports almost all computational and implementation concepts. The only significant **exception** is that the concept of **continuous media** is not supported. Moreover, concepts of **deployment** and target environment are not supported.

The types of eODL are mapped to appropriate types in SDL-2000. Concepts that define aspects of behaviour of types are mapped to automatically implemented SDL agents. Given a complete eODL model, a mapping tool generates a SDL-2000 skeleton. The user has to implement the business logic and is able to use **COs** defined in SDL-2000 as building blocks for a SDL-2000 system.

The mapping aims at supporting concepts as completely as possible even at the cost of producing somewhat complicated structures in SDL-2000. For instance, multiple inheritance of **COs** is supported at the cost of more complex structures and a less efficient behaviour, because SDL-2000 does not support multiple inheritance for agent types.

C.2 The package eodl

The mapping from ITU-eODL to SDL-2000 defines the SDL package `eodl`. It contains definitions of **data types** that are used by models generated from eODL models. Types referred to as predefined are defined in package `eodl`. See C.10 for a complete listing of package `eodl`.

The textual notation of eODL inherits its lexical rules from OMG CORBA-IDL. In CORBA-IDL, unqualified identifiers begin with an alphabetic character followed by any number of alphabetic characters, digits or underscores, where alphabetic characters are English characters 'A' to 'Z' in both uppercase and lowercase. Moreover, identifiers in eODL are case-insensitive.

According to the lexical rules of SDL-2000, all (unqualified) eODL identifiers are SDL-2000 identifiers.

Qualified identifiers are dealt with in clause C.4 (Scoped names).

C.3 Structure

Each ITU-eODL file is mapped onto two SDL packages:

- `<name>_interface` (referred to as "interface package"); and
- `<name>_definition` (referred to as "definition package").

where `<name>` is the name of the ITU-eODL specification (for instance, the file name), as depicted in Figure C.1.

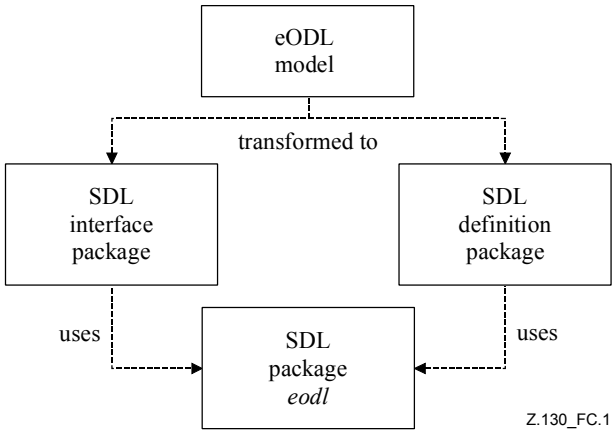


Figure C.1/Z.130 – Transformation structure from an eODL model to SDL

The interface package contains all information that is relevant for both the client and the server side of the system. In details, these are:

- data type definitions;
- constant definitions; and
- interface definitions representing regular **interfaces** and **interfaces** of **COs**.

The definition package contains skeletons for the server sides of the system. In fact, these are:

- block types representing **CO types**; and
- process types representing **artefacts**.

C.4 Scoped names

Qualification is a concept that exists in both eODL and SDL-2000. Thus, the mapping is canonical: qualified names in eODL are mapped onto qualified names in SDL.

C.5 Mapping of computational concepts

C.5.1 Modules

An eODL module is a container for all other eODL elements and opens a namespace. This concept is mapped onto the package concept of SDL.

The SDL package an eODL module is mapped onto may be contained either in the SDL interface package or in the SDL definition package or in both packages, depending on the entities that are enclosed in the eODL module.

C.5.2 Type Definitions

The eODL `typedef` construct assigns a (different) name to a given type. The `typedef` construct is mapped onto the `syntype` construct of SDL.

C.5.3 Predefined Data Types

C.5.3.1 Data Types for Integer Numbers

The eODL data types `unsigned short`, `unsigned long` and `unsigned long long` are mapped onto the following SDL sorts. These sorts are defined in the package `eodl`.

- The eODL type `unsigned short` is mapped onto an `Integer` sort `ushort` that ranges from 0 to $2^{16} - 1$.
- The eODL type `unsigned long` is mapped onto an `Integer` sort `ulong` that ranges from 0 to $2^{32} - 1$.
- The eODL type `unsigned long long` is mapped onto an `Integer` sort `ulong_long` that ranges from 0 to $2^{64} - 1$.

The eODL data types `signed short`, `signed long` and `signed long long` are mapped onto the following SDL sorts.

- The eODL type `signed short` is mapped onto an `Integer` sort `short` that ranges from -2^{15} to $2^{15} - 1$.
- The eODL type `signed long` is mapped onto an `Integer` sort `long` that ranges from -2^{31} to $2^{31} - 1$.
- The eODL type `signed long long` is mapped onto an `Integer` sort `long_long` that ranges from -2^{63} to $2^{63} - 1$.

C.5.3.2 Data Types for floating point numbers

The eODL floating point types `float`, `double` and `long double` are mapped onto the predefined SDL sort `Real`. Note that `Real` is not IEEE 754 [15] complaint according to ITU-T Rec. Z.100, whereas eODL floating point types are.

C.5.3.3 Data Types for characters

The eODL type `char` is mapped onto the predefined SDL sort `Character`. The eODL type `wchar` is mapped onto the predefined SDL sort `Natural`.

C.5.3.4 Data Type boolean

The eODL type `boolean` is mapped onto SDL sort `Boolean` and the eODL `boolean` constants `TRUE` and `FALSE` are mapped onto the `Boolean` literals `true` and `false`, respectively.

C.5.3.5 Data Type octet

The eODL type `octet` is mapped onto SDL sort `Octet`.

C.5.3.6 Data Type any

The eODL type `any` is mapped onto SDL sort `Any`. It should be noted that the semantic of any in eODL is the same as the semantics of any in SDL (see OMG IDL 2.4.2 section 3.10.1.7).

C.5.3.7 Type identification using the type attribute TypeCode

The attribute `typeCode` of an instance of *IDLType* in the **metamodel** of eODL is not mapped onto SDL. However, the type `TypeCode` is mapped onto SDL sort `TypeCode` in package SDL. The sort `TypeCode` is an abstract data type. This means that there are no values of this data type according to this mapping. It does not restrict implementations to use a derived concrete data type, though.

C.5.4 Constructed Data Types

C.5.4.1 Enumerations

Enumerations in eODL are mapped onto a SDL value data type that contains literals only.

C.5.4.2 Structures

Structures in eODL are mapped onto SDL value data types with a public structure data type constructor. Members of the eODL structure are fields of the SDL data type.

C.5.4.3 Unions

Unions in eODL are mapped onto SDL nested value data types. The outer value data type in SDL has a public structure data type constructor with two fields:

1) a field `tag` that represents the discriminator of the eODL-union; and

2) a field `union` that represents the union itself.

The field union is of value data type `<name-of-eODL-union>_union`. This type is declared within the scope of the outer SDL union type. It has a public choice data type constructor and its choice list represents the members of the eODL union.

C.5.4.4 Arrays

Arrays in eODL are mapped onto a SDL sort that inherits from predefined SDL sort `Vector`.

Multidimensional arrays are mapped onto a SDL value data type that supports the operators `Make`, `Modify` and `Extract`.

C.5.5 Value Types

Value types in eODL are mapped onto SDL object data types with a structure data type constructor. All data members of the eODL value type are fields of the SDL data types.

If an eODL value type is declared abstract, the according SDL object data is declared abstract, too. It has no data type constructor.

Single inheritance of value types in eODL is mapped onto single inheritance of data types in SDL. Multiple inheritance is allowed for abstract value types in eODL. In SDL, this is accomplished by copying the operation declaration from the base data types to the derived data type.

Value types in eODL can have factories (initialization elements). These are mapped onto operators in SDL that return a value of the data type. If exactly one factory is declared, the `Make` operator is automatically implemented by calling this factory. Otherwise, it is not exactly implemented.

If the eODL value type supports an interface, the **operations** of that interface become operations of the SDL data type. Attributes in a supported interface are mapped onto a get/set-pair of operations and a field carrying the attribute.

Boxed value types are mapped in the same manner as (concrete) value types.

C.5.6 Parameterized Data Types

C.5.6.1 Sequences

Sequences in eODL are mapped onto SDL sort `Vector` if bounded or onto SDL sort `Array` otherwise.

C.5.6.2 Character Strings

The eODL type `string` is mapped to the predefined SDL sort `Charstring`.

The eODL type `wstring` is mapped to the SDL sort `wstring` in package `eodl`. This sort inherits from `String<Natural>`. It also adds an operation from `Charstring` to convert a `Charstring` value to `wstring` value.

C.5.6.3 Fixed-point Numbers

The SDL package `eodl` contains a type `fixedpt` that is parameterized by a width and a scale **parameter**. Both of these are of sort `Natural`. The semantics of the parameters are equal to the semantics of the parameters in eODL.

The type `fixedpt` defines operators "+","–","*","/" that adds, subtracts, multiplies and divides two values of type `fixedpt` and returns a value of type `fixedpt`. Moreover, the operator "=" compares to `fixedpt` values and returns a `Boolean` value: true, if both operands represent the same number and false if both operands represent different numbers.

To convert a `fixedpt` value to a `Real` value, there is a method `toReal`. It is possible to construct a `fixedpt` value from a `Real` value using a `Make` operator.

C.5.7 Constants, Data Type Literals and Constant Expressions

C.5.7.1 Constants

eODL constants are mapped onto SDL synonyms.

C.5.7.2 Data Type Literals

This clause lists literals of data types that are not mapped onto predefined SDL sorts.

C.5.7.2.1 Literals of Integer Types

Decimal and hexadecimal integer literals are supported in the SDL mapping: decimal literals are supported by SDL `Integer` sort and hexadecimal integer literals can be written as literals of SDL type `Bitstring` and then be converted to `Integer`. Octal integer literals have to be converted to either decimal or hexadecimal literals.

C.5.7.2.2 Literals of character types

A literal of eODL type `char` is mapped such that it forms a legal literal of the predefined SDL sort `Character`. If the eODL literal is a character with a value greater than 127, it cannot be mapped. It is recommended to use the `wchar` type instead.

A literal of eODL type `wchar` is mapped such that it forms a legal `Natural` literal.

C.5.7.2.3 Literals of character strings

A literal of eODL type `string` is mapped such that it forms a legal literal of the predefined SDL sort `Charstring`. If the eODL literal contains a character with a value greater than 127, it cannot be mapped. It is recommended to use the `wstring` type instead.

A literal of eODL type `wstring` is mapped such that it forms a legal literal of SDL sort `wchar`. This includes the possibility to convert a `Charstring` literal into a SDL `wstring` value.

C.5.7.2.4 Literals of fixed point number type

Fixed point values can only be created by converting real numbers.

C.5.7.3 Constant Expressions

Constant expressions in eODL are mapped to constant expressions in SDL. Both expressions have to represent the same values.

C.5.8 Signal Types

Signal types are mapped onto value data types.

C.5.9 Exceptions

Both ITU-eODL and SDL support **exceptions**. The only difference is that SDL **exceptions** do not name **exception** members. **Exceptions** are defined in the interface package.

C.5.10 Interfaces and Interaction Elements

C.5.10.1 Interfaces

An eODL interface groups:

- **operations**;
- **signal** flows;
- **continuous media** interactions (streams);
- attributes.

Continuous media interactions are not mapped onto SDL.

An eODL interface `I` is mapped onto SDL interfaces `exported_I` and `imported_I`. These interfaces are defined within a SDL package `I` in the interface package. The interface `exported_I` contains

- all **operations**; and
- consumed **signal**.

whereas the interface `imported_I` contains produced **signals**.

This mapping is such that the `exported_<interface-name>` interface type contains everything an interface's client needs to invoke a service that the interface represents.

This mapping is detailed in the following clauses.

C.5.10.2 Operational interaction elements

An eODL **operation** is mapped onto a remote procedure that is declared in SDL in the interface `exported_<interface name>`. Both concepts support at most one return type, parameters that can be in, out and inout as well as **exceptions**. Instances of the concept "context" are not mapped.

C.5.10.3 Signal interaction elements

If an eODL interface declares an **interaction element** for consumption of a **signal** `A`, the corresponding SDL interface `exported_<interface-name>` declares the use of signal `A`.

If an eODL interface declares an **interaction element** for production of a **signal** `A`, the corresponding SDL interface `imported_<interface-name>` declares the use of signal `A`.

C.5.10.4 Attribute interaction elements

Attributes in eODL are mapped onto a pair of set/get remote procedures in the `exported_<interface-name>` SDL interface. If the attribute is read-only, the set remote procedure is not generated.

C.5.10.5 Interface References

A SDL interface type implicitly defines a special sort of `PId`. An instance of this `PId` sort serves as an interface reference. Moreover, this concept provides type safety: a client that has got such a `PId` can send signals to and invoke remote procedures on an agent that implements this interface. The SDL interface type that serves as the interface reference type is `exported_<interface name>`.

C.5.10.6 Inheritance

Both ITU-eODL and SDL support multiple inheritance of interfaces. So, the mapping is canonical.

C.5.11 Computational Objects

In eODL, **COs** capsule state and behaviour. They provide interfaces (as interface references) to the environment and may use interfaces (references) of other **COs** themselves.

In SDL, a **CO** is a process type agent. Every **CO** has three gates: one incoming called `initial`, one called `provides` and one called `uses`. These gates support several interfaces as discussed in the following clauses. All three gates are connected to the environment of the process because they constitute the interface between **CO** and environment.

The **CO** process type itself is defined in the definition package.

To create **CO** processes, a **CO** factory is defined. The **CO** factory is represented as a process type defined in the same scope as the **CO** process type. Both the **CO type** itself and its factory are process types defined within an agent type of kind block. This block is called `SDL component`. Note that `SDL component` as defined in this mapping is a concept of the computational view whereas *component* as defined in this Recommendation is a **deployment** view concept. A SDL component represents a block type with a well-defined interface.

C.5.11.1 The interface of a CO

Apart from providing and using interfaces, a **CO type** exposes its own interface. This interface is made up of three components: the user-defined interface, the implicit component identifying interface and the configuration interface.

The **metamodel** defines a **CO type** as an interface kind and restricts the contained **interaction elements** to instances of *AttributeDef*. When regarding the **CO type** as an interface only, this is called "user-defined interface".

To identify **COs**, every **CO type** has an implicit read-only attribute key of predefined type `ComponentKey`. This attribute is the only **interaction element** in the predefined interface `ComponentBase`.

The interface `ComponentBase` defines access procedure to the key of a **CO**. The key of a **CO** implements its identity and needs to be unique at least with respect to the SDL component that the **CO** is contained in. In package `eodl` there is declared an external procedure `compute_co_key` that returns such a unique key. An implementation of this mapping has either to provide an implementation of that procedure or to generate code that computes a unique key in another way.

The configuration interface defines procedures to support **port** operations. This interface is further detailed in the clause on the mapping of ports. The configuration interface inherits from the predefined interface `ConfigBase`. `ConfigBase` declares generic **port** operations.

All these interfaces are defined in the interface package. They are contained within a package of the name of the **CO**. The user-defined interface is called `<CO-name>_attributes`. It inherits from the predefined interface `ComponentBase`. The configuration interface is called `<CO-name>_config` and inherits from the predefined interface `ConfigBase`. Finally, the interface `<CO-name>` is defined that simply inherits from `<CO-name>_attributes` and `<CO-name>_config`.

The interface `<CO-name>` is supported by the gate `initial`. A channel from the environment to the **CO** process type is defined.

C.5.11.2 Supported interfaces

The `provides` gate references all interfaces that the **CO supports**. Through this gate, clients can use the **CO**. The SDL `interface exported_<name>` is supported in the direction from the environment to the process type and the SDL interface `imported_<name>` is supported in the direction from the process type to the environment.

C.5.11.3 Required Interfaces

The `uses` gate references all interfaces that the **CO requires** and uses. Through this gate, the **CO** (in the role of a client) can use other **COs**. The SDL interface `imported_<name>` is supported in

the direction from the environment to the process type and the SDL interface `exported_<name>` is supported in the direction from the process type to the environment.

C.5.11.4 Inheritance

Since SDL does not support multiple inheritance, inheritance is realized by delegation. This is realized as follows:

1) The **CO type** user-defined interface and the **CO type** configuration interface inherits from the corresponding interfaces of the super **CO**(s).
2) The gate `provides` supports all interfaces that are supported by the super **CO**(s). It also supports all interfaces supported by the **CO type** itself.
3) The gate `uses` supports all interfaces that are required by the super **CO**(s). It also supports all interfaces required by the **CO type** itself.

For the behaviour aspects of multiple inheritance, see C.7.2.5.

C.5.11.5 CO factories

To every **CO type** there is an associated **CO** factory `<CO-name>_factory`. It implements the factory interface. This interface is defined in a package `<CO-name>_factory` that in turn is defined in the same package where the interfaces of the **CO** are defined. The factory interface inherits the predefined interface `CoFactoryBase`.

The interface `CoFactoryBase` contains the following procedures. The procedure `get_co_type` returns the fully qualified name of the **CO type**. Qualification character is the decimal point. The procedure `generic_create` instantiates the associated **CO type**. The procedure `list_cos` returns a list of values of `ComponentKey` of instantiated **COs**. The procedure `resolve_co` takes a `ComponentKey` value and returns the associated **CO**.

The factory interface declares a procedure `create_<CO-name>` that creates the associated **CO type**. The factory agent has a gate `factory` that supports the `CoFactoryBase` interface in incoming direction.

C.5.11.6 Encapsulation of CO type and CO factory – SDL component

Both the **CO type** and **CO** factory type are defined in a block type `<CO-name>_CO`. This block type is called SDL component. In every SDL component, there is an instance set `factory` of type `<CO-name>_factory` as well as an instance set `cos` of type `<CO-name>`. The instance set `factory` contains exactly one instance. The instance set `cos` contains initially no instance and no restriction regarding the maximum number of instances. The `initial`, `provides` and `uses` gates of the **CO type** are duplicated by the SDL component as well as the `factory` gate of the factory type and those gates are connected by channels.

The SDL component is defined in the definition package. It represents a **CO type**.

C.6 Mapping of configuration view concepts

C.6.1 Provided ports

The concept of **provided ports** in eODL is a mechanism to hand out interface references that are provided by a **CO** to the clients of this **CO**.

This concept is mapped onto a set of remote procedures that are declared in the configuration interface of the **CO**.

A **provided port** `foo` of type `bar` is mapped onto the remote procedure `provide_foo` that returns a reference (`PId`) to `bar`. If `foo` is of attribute **single**, every call to `provide_foo` is required to return

the same PId. If foo is of attribute **multiple**, the user has to implement the semantic himself (see C.7.4.3 on **port** management).

The configuration interface inherits from the predefined interface ConfigBase. This interface declares a procedure provide. It takes a string as an argument. The actual **parameter** in the procedure call designates a **port**. If this port exists, a reference is returned in the form of a PId. If the **port** does not exist, a NoSuchPort **exception** is raised. The **exception** NoSuchPort is predefined.

C.6.2 Used ports

The concept of **used port** in eODL is a mechanism that enables a **CO** to store interface references of other **COs**.

This concept is mapped onto a set of remote procedures that are declared in the configuration interface of the **CO**.

A **used port** foo of type bar is mapped onto the remote procedure link_foo that takes a reference to bar as **parameter**. If the **port** is of attribute **single** and there is already a reference stored at this **port**, the predefined **exception** AlreadyConnected is raised. Moreover, a remote procedure unlink_foo is declared that removes the stored reference from **port** foo. If there is no reference stored at foo, the predefined **exception** NotConnected is raised. If the **port** is of attribute **multiple**, a sequence of references is stored. The **exception** is AlreadyConnected never raised.

The predefined interface ConfigBase that the configuration interface is inheriting from declares a procedure link and a procedure unlink. These procedures can be used in a generic way to store or delete a reference at a **used port**. As their counterpart for **provided ports**, they take a string argument that designates the **port** name. Again, if the **port** with the designated name does not exist, a NoSuchPort **exception** is raised. The generic connect procedure takes a PId as a second argument and stores it at the designated **port** if possible or raises an AlreadyConnected otherwise. It uses the same semantics as the **port** specific connect procedure to decide whether storing the reference is possible. The generic disconnect procedure raises a NotConnected **exception** if there is no reference stored at the designated **port**.

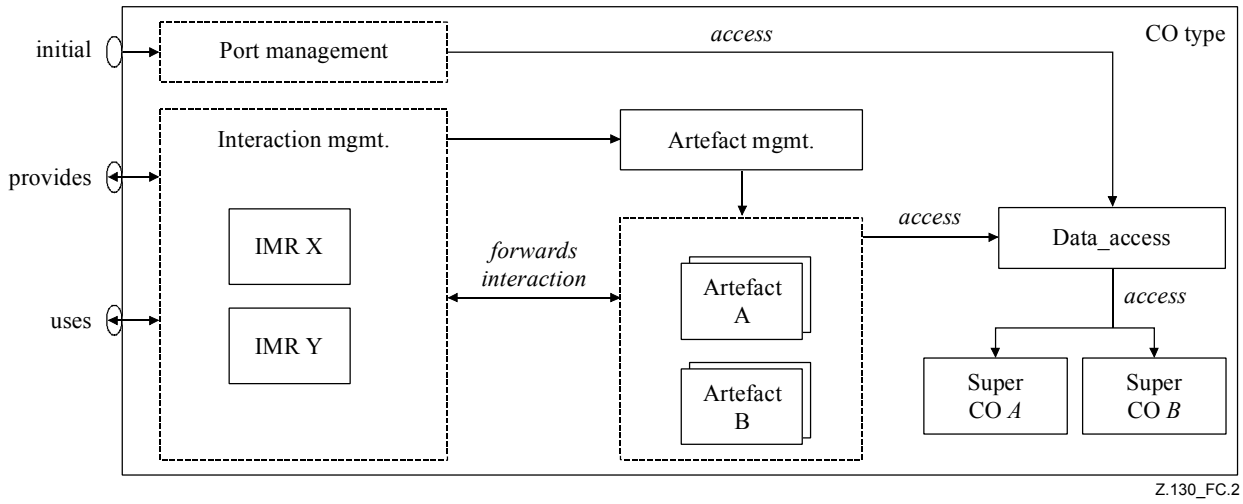
C.6.3 A SDL naming service

For a **CO** to dynamically discover other SDL components, a naming service is defined. The naming service is implemented by the process type SDL_Component_Register_Type in package eodl. Each SDL system derived from an eODL model is required to have an instance set SDL_Component_Registry that contains exactly one instance of type SDL_Component_Register_Type.

As soon as a SDL component is instantiated, it registers itself using the exported procedure register_SDLComponent. Any **CO** can query the naming service using query_SDLComponent. The key to look up a SDL component is the fully qualified name using the period character as qualification character. The query procedure returns a reference to an SDL component instance of the requested type. Using this reference, any client can request the factory of the SDL component for **COs**.

C.7 Mapping of implementation concepts

Figure C.2 depicts the internal structure of a SDL process representing a **CO type** in the form of an overview with different optional SDL representations.



IMR Interaction Management Representation

Figure C.2/Z.130 – SDL process representing CO type

A rectangle represents a SDL process. A dotted rectangle represents a concept rather than a process. For instance, "interaction management" represents the concept of interaction management and contains concrete interaction management processes as SDL processes.

In Figure C.3, a concrete example of a CO process type is presented in SDL graphical notation. The procedures `get_key`, `provide_SamplePort` and `provide` make up the port management. The processes `interaction_interfaceX` and `interaction_interfaceY` correspond to "IMR X" and "IMR Y" in Figure C.2. The process instance sets `A` and `B` are instances of **artefacts**. The instance set `base` correspond to the box "Super CO A" in Figure C.2.

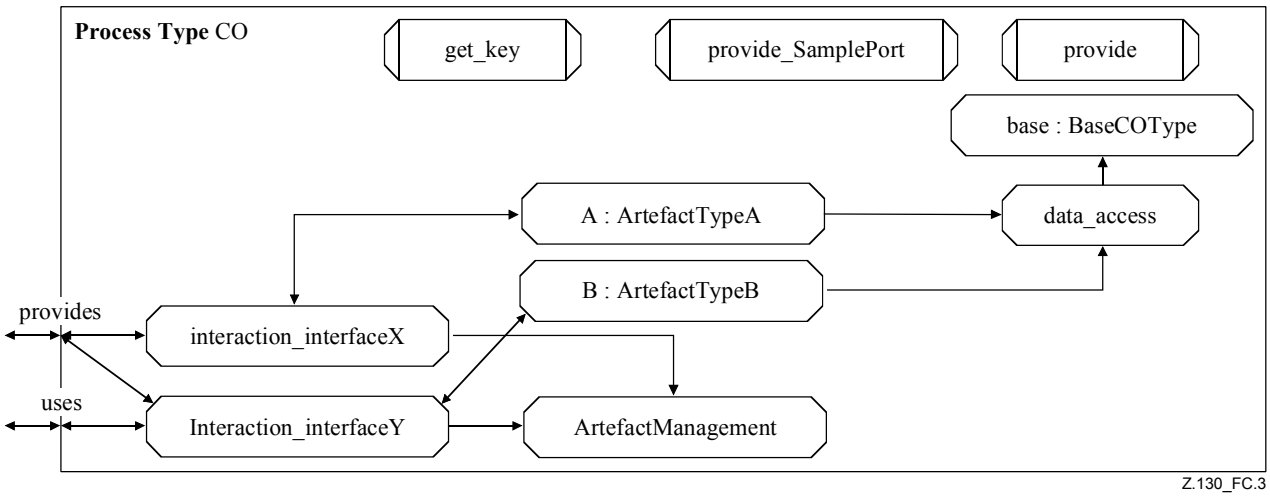


Figure C.3/Z.130 – Example of a CO process type

C.7.1 Data Access

Data of a CO is stored in a process `data_access` that implements get/set procedures to allow artefacts access to these data. The data consists of references used by port management and interaction management.

To provide a typed access to that data, a package `<CO-Name>_data` is declared in the definition package. This data package contains an interface `internal_data` that declares all get/set procedures.

C.7.2 Artefacts and implementation elements

Artefacts are programming language constructs that contain **implementation elements**. In SDL-2000 they are mapped onto referenced process types that are defined in the definition packages. **Artefacts** are instantiated as an instance set within the **CO** process type it implements.

To access data in a **CO**, a channel from **artefact** instance set to the `data_access` process is defined. It carries all procedure calls to the `data_access` process.

Implementation elements associate **artefacts** and **interaction elements** that the **artefact** implements. They have no representation in SDL.

The implementation of an **interaction element** depends on its implementation case. There are two implementation cases:

- supply case;
- use case.

Table C.1 gives the semantics of the **interaction element** kinds and implementation cases.

Table C.1/Z.130 – Semantics of interaction element kinds and implementation cases

Type of interaction element in design model	Case definition of implementation element in design model	Semantic of implementation elements
operation/attribute	supply	Implementation of **operation** behaviour/supply access **operations** to attribute
operation/attribute	use	Call to operation explicit possible
consume	supply	Implementation of signal consumption
consume	use	Implementation of signal sending
produce	supply	Implementation of signal sending
produce	use	Implementation of signal consumption

Similar to the mapping of interfaces as described in C.5.10 about interfaces and interaction elements, each eODL interface is additionally mapped in the definition package onto a package of the same name as the interface containing two SDL interfaces: `exported_<Interface-name>` and `imported_<Interface-name>`. The mapping of the **interaction elements** is exactly such as described in C.5.10 except for procedure and signal. Every procedure and every signal bear an additional formal **parameter** of type `PId`. This **parameter** is used to carry sender respectively receiver information. See C.7.4 for further information. The signal types that these interfaces refer to (bearing an additional formal **parameter** of type `PId`) are defined in the definition package.

The following clauses contain further details on implementing **interaction elements**.

C.7.2.1 Implementing operation

In order to implement an **operational interaction element** of an interface, the **artefact** has to contain an exported procedure that implements the procedure defined in the SDL interface `exported_<Interface-Name>` of the corresponding interface in the definition package (see last clause). It therefore implements a procedure that contains an additional **parameter** of type `PId`. The **artefact** can use this **parameter** to get information about the original sender of the procedure call.

C.7.2.2 Calling an operation from an artefact

Operation calls to other **COs** are realized as follows: the **artefact** sends a procedure call to that interaction management representation which implements the `imported_<interface-name>` of the interface that contains the operation to call. This procedure is defined in the definition package and contains an additional formal **parameter** of type `PId`. The actual `PId` **parameter** is used to designate the receiver of the procedure call. The interaction management representation is responsible for forwarding the procedure call to its receiver.

C.7.2.3 Sending a signal

Similar to the calling of a procedure, an **artefact** does not send a signal directly to its receiver, but to the interaction management representation. The signal contains an additional formal **parameter** that designates the receiver of the signal and is defined in the definition package. The interaction management representation is responsible for forwarding the signal to its receiver.

C.7.2.4 Consuming a signal

In order to implement the consumption of a signal, the **artefact** needs to implement a signal handler that accepts the corresponding signal defined in the definition package.

C.7.2.5 Inheritance of artefacts

Multiple inheritance is allowed for **artefacts**. Since in SDL there is no multiple inheritance allowed for agent types, this is realized by delegation. Base **artefacts** are contained in the derived **artefact** and the corresponding gates of each base **artefact** and the derived **artefact** are connected by a channel. All procedure calls and signals that go to **implementation elements** that are not redefined are directly passed from the environment of the derived **artefact** to the appropriate base **artefact**. If a certain **implementation element** is being redefined in the derived **artefact**, the redefined **implementation element** is defined in the derived **artefact**.

C.7.3 Artefact management and instantiation pattern

Artefact management is responsible for creating and managing instances of an **artefact**. Within a **CO**, there is an instance set `artefact_<CO-name>` for every **artefact** that implements the **CO**. However, the instantiation pattern that is annotated in the model defines what **artefact** instance is used in an interaction. The **artefact** management implements the instantiation pattern.

The **artefact** management is realized as a process `artefactmanagement` contained in the **CO** process type. The instance set of that process contains exactly one instance. For every **artefact** type that implements the given **CO**, there is an exported remote procedure `get_artefact_<artefact-name>` that returns a reference to an instance of that **artefact** type. These procedures are automatically implemented. Their implementation depends upon the instantiation pattern to be used:

- *Artefact by request*: During every procedure call, a new instance is created and a reference to it is returned;
- *Artefact-Pool*: A limited number of instances (a "pool") is created and a reference to one of these instances is returned;
- *Singleton*: There is only one instance of the **artefact**, and a reference to it is returned;
- *Userdefined*: Since the semantics is defined by the user, the procedure cannot be automatically implemented. Instead, a referenced procedure is declared and the user has to implement it himself.

C.7.4 Interaction management representation

In the mapping to SDL, the interaction management representation acts as a proxy between the **implementation elements** and the environment of a **CO**. Each **interaction element** representation handles both incoming and outgoing interactions (referred to the **CO**).

Each interface that is required or supported by a **CO** is represented by a process in the scope of the **CO** process type. This process implements each **interaction element** either by forwarding the interaction request:

- from the environment of the **CO** to the appropriate **implementation element**, thereby respecting the **artefact** instantiating pattern by using the **artefact** management representation; or
- from the **implementation element** to the environment of the **CO**.

There is one and only one process instance per interface. However, if there is a **multiple port** of this type, then there is an exception to this rule. In this case, there is a certain number of instances. The concrete number is subject to the implementation. Moreover, for each interaction management process, there is an implicit **CO**-internal variable `<process-name>_reference` that holds the `PId` of that process. If there is more than one instance reference to hold, a string of `PIds` is used.

C.7.4.1 Interaction management representation implementing interaction from CO to environment

For interactions to the environment the interaction management representation implements:

- all **operation** calls (only for required interfaces); and
- all signals that might be sent.

It supports the interface `imported_<interface-name>` of the definition package in the incoming direction (from **artefacts**) and `imported_<interface-name>` of the interface definition in the outgoing direction (to environment).

An **operation** (more specifically, the use of an **operation**) is implemented as follows:

1) The value of `sender` is saved to a temporary variable.
2) The reference to an **artefact** instance is acquired by calling the appropriate procedure of the **artefact** management representation.
3) The procedure is called (the saved value of `sender` is added to the list of parameters) with the reference to the **artefact** instance as destination.
4) If the **operation** has **exceptions** declared, those **exceptions** have to be catched and raised again.

A sending of a signal is implemented as follows: when the **interaction element** representation receives the specified signal (with an additional **parameter** that specifies the destination), it sends the specified signal (without the additional **parameter**) to the destination.

C.7.4.2 Interaction management representation from environment to CO

For interactions in this direction, the interaction management representation implements:

- all **operations** declared in the interface; and
- all produced and consumed **signals**.

It supports the interface `exported_<interface-name>` of interface package in the incoming direction (from environment) and `exported_<interface-name>` of definition package in the outgoing direction (to **artefacts**).

An **operation** (more specifically, the supply of an **operation** to the environment) is implemented as follows. When the procedure call is received, an **artefact** instance reference is acquired by calling

artefact management. Then, a procedure call to that instance is made, supplying the value of the `sender` variable as an additional **parameter**. If the **operation** has a return **parameter**, the return value is returned to the sender. If an **operation** is raised, the interaction management representation is raised again.

Consuming a **signal** is implemented as follows: when the **interaction element** representation receives the signal from the environment, it sends the corresponding signal (with an additional **parameter** that contains the value of the `sender` variable) to the **artefact**.

C.7.4.3 Port management

Port management is responsible for:

- creating **interaction element** representations;
- managing references to interfaces;
- implementing **port**-specific accessor **operations**; and
- implementing generic **port** accessor operations.

All these responsibilities are automatically implemented.

C.7.4.4 Port management representation

The port management is implemented by several exported procedures and the start transition of the state machine of the **CO type** process.

C.7.4.5 Creating interaction management representations

The port management is responsible for creating processes representing interaction management. This is realized in the start transition of the **CO type** process. After creation of the interaction management process, the port management stores the references to each process in the data access process using the variable `interaction_<interface-name>`.

C.7.4.6 Managing references to interfaces

For each **port**, there is an implicit **CO**-local variable `port_<port-name>`. The type of this variable is either the reference type of the interface typing the port (when **port** attribute is **single**) or string-of-PId (when **port** attribute is **multiple**). **Port** operations directly manipulate these internal variables.

C.7.4.7 Implementing port-specific accessor operations

The particular operations are implemented as follows.

Single provided port: This operation returns the reference that is stored in the internal variable.

Multiple provided port: One reference out of a set of references has to be chosen. The procedure `choose_provide_<port-name>` (defined within the port management process) is called to make this decision. This procedure has to be implemented by the user and is therefore referenced.

Single used port: The operation `link` stores a given reference in the data access process using the variable `port_<port-name>`. If there is already a reference stored there, an **exception** `AlreadyConnected` is raised. The operation `unlink` assigns the `Null` value to the **CO**-local variable. If the `Null` value is already assigned to it, the operation raises an **exception** `NotConnected`.

Multiple used port: The operation `link` stores a given reference in a sequence of references. To determine where exactly to place the reference within the sequence, the user has to implement the procedure `choose_link_<port-name>` (defined within the port management process) that takes the reference and has to store it somewhere in the sequence. Likewise, the disconnect operation calls `choose_link_<port-name>` to remove an appropriate reference.

C.8 Omitting automatically generated behaviour

The eODL-SDL mapping presented so far is driven by the idea that the user only wants to implement the business logic. However, if the user wants to generate production code out of SDL, he or she might want to avoid automatically generated code and want to implement it on their own.

To enable this, the mapping provides the option to omit automatically generated code. In detail, this means that the following entities are not generated:

- **artefact** management and **artefact** instance sets;
- interaction management representations; and
- port management.

In short, the **CO** process type does not contain any entities. The user can implement the **CO** process type in any way he or she wants to.

C.9 Not mapped eODL concepts

The following eODL concepts are not mapped onto SDL-2000:

- any-type **runtime** identification;
- **continuous media** interaction;
- **software components**;
- **assemblies**;
- constraints and properties;
- target environment concepts;
- deployment plan.

C.10 Predefined eodl package

This is the complete contents of the package **`eodl`**.

```
package eODL;
  syntype unsigned_short = Integer constants (0:65535);
  endsyntype;

  syntype unsigned_long = Integer constants (0:4294967295);
  endsyntype;

  syntype unsigned_long_long = Integer constants (0:18446744073709551615);
  endsyntype;

  syntype short = Integer constants (-32768:32767);
  endsyntype;

  syntype long = Integer constants (-2147483648:2147483647);
  endsyntype;

  syntype long_long = Integer constants
    (9223372036854775808:9223372036854775807);
  endsyntype;

  syntype char = Character endsyntype;
  syntype wchar = Natural endsyntype;

  syntype float = Real endsyntype;
  syntype double = Real endsyntype;
  syntype long_double = Real endsyntype;
```

```
value type wstring
  inherits String < wchar >;
endvalue type wstring;

value type wstring_bounded < synonym length Natural >
   inherits Vector < wchar, length>;
endvalue type wstring_bounded;

abstract value type TypeCode;
endvalue type;

value type fixedpt < synonym Width Natural; synonym scale Natural >;
  struct
    private unscaled_int Integer;
  operators
    Make ( Real ) -> this fixedpt;
    Make ( Integer ) -> this fixedpt;
     "+"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "-"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "*"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "/"  ( this fixedpt, this fixedpt ) -> this fixedpt;
     "="  ( this fixedpt, this fixedpt ) -> Boolean;
     ">"  ( this fixedpt, this fixedpt ) -> Boolean;
  methods
    toReal -> Real;

  OPERATOR Make (r Real ) -> this fixedpt  {
    DCL retVal this fixedpt;
    r := r * power(10,scale);
    retVal.unscaled_int := fix(r);
    return retVal;
  }
  OPERATOR Make (n Integer ) -> this fixedpt  {
    DCL retVal this fixedpt;
    retVal.unscaled_int := n * power(10,scale);
    return retVal;
  }
  OPERATOR  "+"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt;
    retVal.unscaled_int := a.unscaled_int + b.unscaled_int;
    return retVal;
  }
  OPERATOR  "-"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt;
    retVal.unscaled_int := a.unscaled_int - b.unscaled_int;
    return retVal;
  }
  OPERATOR  "*"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt,
        t Real;
    t := float(a.unscaled_int * b.unscaled_int);
     t := t / float(power(10,2*scale));
    retVal.unscaled_int := Make( t );
    return retVal;
  }
  OPERATOR  "/"  (a this fixedpt, b this fixedpt ) -> this fixedpt {
    DCL retVal this fixedpt,
         t Real;
    t := float(a.unscaled_int)/float(a.unscaled_int);
    retVal.unscaled_int := Make( t );
    return retVal;
  }
```

```
  OPERATOR  "="  (a this fixedpt, b this fixedpt ) -> Boolean {
    return a.unscaled_int = b.unscaled_int;
  }
  OPERATOR  ">"  (a this fixedpt, b this fixedpt ) -> Boolean {
    return a.unscaled_int > b.unscaled_int;
  }
  METHOD toReal -> Real {
    return float(unscaled_int)/float(power(10,scale));
  }
endvalue type;

package ComponentModel;
  value type ComponentKey;
     struct
       the_key string;
     methods
       virtual equal (this ComponentKey) -> Boolean;
  endvalue type;
  interface ComponentBase;
    procedure get_key -> ComponentKey;
  endinterface ComponentBase;

  procedure generate_CO_key -> ComponentKey external;

  value type ComponentKeySeq
    inherits String < ComponentKey >;
  endvalue type;
  interface CoFactoryBase;
    procedure get_co_typ -> string;
    procedure generic_create -> ComponentBase;
    procedure resolve_CO (ComponentKey) -> ComponentBase;
    procedure list_cos -> ComponentKeySeq;
  endinterface CoFactoryBase;

  exception NotConnected;
  interface ConfigBase;
    procedure provide(in string) -> PId
         raise NoSuchPort;
    procedure link(in string, in PId)
         raise AlreadyConnected, NoSuchPort;
    procedure unlink(in string, in ComponentBase)
         raise NotConnected, NoSuchPort;
  endinterface ConfigBase;

endpackage ComponentModel;

interface SDLComponent_Registry_IF;
  procedure register_SDLComponent(in string, in PId);
  procedure query_SDLComponent(in string) -> PId;
endinterface;

process type SDLComponent_Registry_Type;
  gate registry in with SDLComponent_Registry_IF;
  channel nodelay
    from env  to this via registry;
  endchannel;

  value type registry_store
    inherits Array < string, Pid >;
  endvalue type registry_store;

  dcl store registry_store := Make;

  exported as <<package eodl>>register_SDLComponent
```

```
    procedure register_SDLComponent(in key string, in item Pid);
      start;
      task registry_store := Modify(store,key,item);
      return;
    endprocedure;

    exported as <<package eodl>>query_SDLComponent
    procedure register_SDLComponent(in key string) ->  Pid
           raise InvalidIndex;
      dcl retval Pid;
      start;
      task retval := Extract(store,key);
      return retval;
    endprocedure;

  endprocess type SDLComponent_Registry_Type;


endpackage eODL;
```

Annex D

eODL metamodel XML representation

The **metamodel** was defined using UML. Its XML representation according to OMG XMI [6] is intended to be read by tools and constitutes Annex D. The actual data is available with the software package "Z.130 Annex D.xml".

NOTE – Z.130 Annex D.xml software package is available for free on the ITU-T formal language database at http://www.itu.int/ITU-T/formal-language/xml/database/itu-t/z/z130/2003/.

Appendix I

Example: Dining Philosophers

I.1 Introduction

The purpose of this appendix is to show an example of how eODL can be used for design, implementation and **deployment** of a distributed system.

The Dining Philosophers problem was first described by Edsger W. Dijkstra in 1965. It is a model and universal method for testing and comparing theories on resource allocation. Dijkstra hoped to use it to help create a layered operating system, by creating a machine which could be considered to be an entirely deterministic automaton.

A configurable number of philosophers (processes) are sitting on a round table; a finite number of forks (resources) are on the table. Philosophers perform actions – thinking, eating and sleeping. They do not need any resources in order to think or sleep, but they need two forks each in order to eat, one for the left hand and one for the right hand. Therefore, before starting to eat, a philosopher tries to get the two forks, which are to be available next to him. This means that two neighbour philosophers cannot eat at the same time.

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

I.2 Description

The problem consists of a finite set of processes which share a finite set of resources, each of which can be used by only one process at a time, thus leading to potential deadlock and lifelock situations.

The finite set of processes, resources and the dynamic interactions between these make up a distributed system. The task is to distribute the implementations of the resources and processes across the target network. Furthermore, resources have to be connected with the processes.

The example scenario includes three different **CO types**:

- Philosopher.
- Fork.
- Observer.

The following steps have to be performed in order to design, implement and deploy the example:

Design phase

- Definition of a model of the example elements, comprising **CO types**, ports and interfaces.
- Definition of a model of the implementation structure.

Implementation phase

- Implementation of the **artefacts** according to the model (provide the business logic).
- Generation of **software components** according to the model.
- Definition of a model of the initial system structure (**initial configuration**) by the definition of an assembly.
- Packaging of the **software components** and their related model information in order to allow shipment of the implementation to customers.

Integration phase

- Delivery of the package to a customer.
- Modelling of the target environment of the customers premise.
- Determination of a proper assignment of **software components** contained in the package to the target environment.
- Installation of assigned **software components** on identified target nodes.
- Establishment of the **initial configuration** by interconnection of all **initial COs** according to the **initial configuration**.

In I.3 the "Dining Philosophers" example is specified with eODL. In the specification, three **CO types** are defined:

- The o_Philosopher object type represents a philosopher.
- The o_Fork object type represents a fork.
- The o_Observer object type represents an observer.

Clause I.4 contains the mapping of the eODL model according to the mapping rules given in Annex C. Only the concepts of computational, configuration and implementation views are mapped as there is no mapping for concepts of the deployment view. The SDL model consists of the two main packages:

- the SDL interface package phil_interface; and
- the SDL definition package phil_definition.

I.3 Example in eODL

```
module DiningPhilosophers {
     CO o_Philosopher;
     CO o_Fork;

     interface i_Fork;
     interface i_Philosopher;
     interface i_Observer;

     exception ForkNotAvailable {};
     exception NotTheEater {};

     enum e_ForkState {
          UNUSED,
          USED,
          WASHED
     };

     enum e_Pstate {
          EATING,
          THINKING,
          SLEEPING,
          DEAD,
          CREATED,
          HUNGRY
     };

     interface i_Fork {
        void obtain_fork ( in o_Philosopher eater )
        raises ( ForkNotAvailable );
        void release_fork ( in o_Philosopher eater ) raises ( NotTheEater );
     };

     artefact a_ForkImpl {
          obtain_fork implements supply i_Fork::obtain_fork;
          release_fork implements supply i_Fork::release_fork;
     };

     CO o_Fork {
          supports i_Fork;
          provide i_Fork fork;
          implemented by a_ForkImpl with ArtefactPool(2);
     };

     interface i_Philosopher {
          void set_name ( in string name);
     };

     artefact a_PhilosopherImpl {
          set_name_impl implements supply i_Philosopher::set_name;
          pstate_impl implements use i_Observer::pstate;
     };

     CO o_Philosopher {
          implemented by a_PhilosopherImpl with Singleton;
          supports i_Philosopher;
          requires i_Fork, i_Observer;
          use I_observer observer;
          use i_Fork left;
```

```
        use i_Fork right;
    };

    valuetype Pstate {
        public e_PState state;
        public string name;
        public i_Philosopher philosoph;
        factory create (
            in e_PState state,
            in string name,
            ini_Philosopher philo);
    };

    signal PhilosopherState {
        PState carry_pstate;
    };

    interface i_Observer {
        consume PhilosopherState pstate;
    };

    artefact a_Observer {
        pstate_Impl implements supply i_Observer::pstate;
    };

    CO o_Observer {
        implemented by a_Observer with Singleton;
        supports i_Observer;
        provide i_Observer observer;
    };
};
softwarecomponent Philosopher
realizes o_Philosopher, o_Observer
{
    requires {
        property os =  [
                            { name = "WINNT"; version = "4,0,0,0"; },
                            { name = "WIN98"; }
        ];
    };
};

softwarecomponent Fork
realizes o_Fork;
{
    requires {
        property os =  [
                            { name = "WINNT"; version = "4,0,0,0"; },
                            { name = "WIN98"; }
        ];
    };
};

assembly ass1 {
    p (3) : o_Philosopher;
    f1 : o_Fork;
    f2 : o_Fork;
    o : o_Observer;
    connect c1 {
        p.left  = f1.fork;
        p.right = f2.fork;
    };
```

```
     connect c2 { o.observer = p.observer; };
};

environment myenv_1 {
     node n1 {
          property os =  { name = "WINNT"; version = "4,0,0,0"; };
          property memory = 256;
     };

     node n2 {
          property os =  { name = "WINNT"; version = "4,0,0,0"; };
          property memory = 128;
     };

     link l1 { node n1, n2; };
};

installation install1
uses environment myenv_1 {
     Philosopher ->n2;
     Fork ->n1;
};

instantiation instantiate1
uses environment myenv_1
uses assembly ass1 {
     p,  o  -> n2;
     f1, f2 ->  n1;
};

deploy {
     install { install1; };
     instantiate { instantiate1; };
};
```

I.4 Example in SDL-2000

```
use eODL;
/* /--------------------------------\ */
/*     data types and interface       */
/*     needed by clients              */
package phil_interface;

package DiningPhilosophers;

  /* exceptions */
  exception ForkNotAvailable;
  exception NotTheEater;

  /* enumerations */
  value type e_ForkState;
    literals
      UNUSED, USED;
  endvalue type;

  value type e_ForkState;
    literals
      EATING, THINKING, SLEEPING,
      DEAD, CREATED, HUNGRY;
  endvalue type;

  /* interface i_Fork */
  package i_Fork;
```

```
    /* declaration of exported procedures */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher)
             raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher)
               raise NoTheEater;
    endinterface;

     /* declaration of consumed signals */
    interface imported_i_Fork;
         /* no consumed signals declared */
    endinterface;
  endpackage;

  /* definition of CO type o_Fork */
  use i_Fork;
  package o_Fork;

    /* contains attributes defined in CO type o_Fork */
    interface o_Fork_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    /* port operations */
    interface o_Fork_config
        inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      /* provided port "fork" */
      procedure provide_fork -> exported_i_Fork;
    endinterface;

    /* combine config and attributes interfaces */
    interface o_Fork
      inherits o_Fork_attributes, o_Fork_config;
    endinterface;

    /* declaration of interface of CO factory */
    package factory;
      interface o_Fork_factory
       inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
        procedure create_o_Fork -> o_Fork;
      endinterface;
    endpackage;

  endpackage o_Fork;

package i_Philosopher;

    interface exported_i_Philosopher;
      procedure set_name(in string);
    endinterface;

    interface imported_i_Philosopher;
    endinterface;

  endpackage;

  use i_Philosopher;
  use o_Philosopher;
  object type Pstate;
    struct
      public eodl_state e_PState; /* state -> eodl_state ! */
      public name string;
       public philosoph exported_i_Philosopher;
    operators
```

```
       /* create -> eodl_create) */
      eodl_create(e_PState, string, exported_i_Philosopher) -> Pstate;
      make(e_PState, string, exported_i_Philosopher) -> PState;

    operator eodl_create(eodl_state e_PState,
                         name string,
                         philo exported_i_Philosopher) {
      dcl retval Pstate;
       retval.eodl_state := eodl_state;
      retval.name := name;
      retval.philosoph := philo;
       return retval;
    }
    operator make(eodl_state e_PState,
                  name string,
                  philo exported_i_Philosopher) {
       return eold_create(eodl_state,name,philo);
    }
  endobject type;

  use i_Philosopher;
  package o_Philosopher;

    /* contains attributes defined in CO type o_Fork */
    interface o_Philosopher_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase;
    endinterface;

    interface o_Philosopher_config
     inherits <<package eODL/package ComponentModel>>ConfigBase adding;
      procedure link_observer(exported_i_Observer) raise AlreadyConnected;
      procedure link_left(exported_i_Fork) raise AlreadyConnected;
      procedure link_right(exported_i_Fork) raise AlreadyConnected;
      procedure unlink_observer raise NotConnected;
      procedure unlink_left raise NotConnected;
      procedure unlink_right raise NotConnected;
    endinterface;

    interface o_Philosopher
         inherits o_Philosopher_attributes, o_Philosopher_config;
    endinterface;

    use eODL / package ComponentModel;
    package factory;
      interface o_Fork_factory
         inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
          procedure create_o_Philosopher -> o_Philosopher;
      endinterface;
    endpackage factory;

  endpackage;

  signal PhilosopherState(PState);

package i_Observer;

    interface exported_i_Observer;
      use PhilosopherState;
    endinterface;

    interface imported_i_Observer;
    endinterface;

  endpackage;
```

```
  /* CO o_Observer */
  use i_Observer;
  package o_Observer;

    interface o_Observer_attributes
        inherits <<package eODL/package ComponentModel>>ComponentBase adding;
    endinterface;

    interface o_Observer_config
        inherits <<package eODL/package ComponentModel>>ConfigBase adding;
    endinterface;

    interface o_Observer inherits o_Observer_attributes, o_Observer_config;
    endinterface;

    package factory;
      interface o_Observer_factory
       inherits <<package eODL/package ComponentModel>>CoFactoryBase adding;
        procedure create_o_Observer -> o_Observer;
      endinterface;
    endpackage;

  endpackage;

endpackage DiningPhilosophers;

endpackage phil_interface;
/* \--------------------------------/ */

/* /--------------------------------\ */
/*     implementation package         */
use eODL;
package phil_definition;

package DiningPhilosophers;


  /* used to define operations implemented or */
  /* used by artefacts */
  package i_Fork;
    /* operations implemented by artefacts */
    interface exported_i_Fork;
      procedure obtain_fork(in o_Philosopher, in Pid)
             raise ForkNotAvailable;
      procedure release_fork(in o_Philosopher, in Pid)
                raise NoTheEater;
    endinterface;

    /* operations used by artefacts */
    interface imported_i_Fork;
    endinterface;
  endpackage;


  /* state attributes */
  package o_Fork_data;

    interface internal_data;
      procedure get_port_fork -> exported_i_Fork;
      procedure get_interaction_i_Fork -> exported_i_Fork;
      procedure set_port_fork(exported_i_Fork);
      procedure set_interaction_i_Fork(exported_i_Fork);
```

```
  endinterface;
endpackage;

signallist a_ForkImpl_in =
     procedure <<package phil_definition/package DiningPhilosophers/
                   package i_Fork>>obain_fork,
      procedure <<package phil_definition/package DiningPhilosophers/
                  package i_Fork>>release_fork;

/* artefact: referenced definition */
use a_PhilosopherImpl;
use o_Fork_state;
process type a_ForkImpl with
   use (a_ForkImpl_in);;
  referenced;

/* /--------------------------------\ */
/*     SDL component o_Fork_CO        */
use a_PhilosopherImpl;
use o_Fork_state;
use a_ForkImpl;
block type o_Fork_CO;

  /* gate definitions */
  gate factory
    in with <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>o_Fork_factory;
  gate initial
    in with  (<<package phil_interface/
        package DiningPhilosopher/package o_Fork>>o_Fork);
  gate provides
    in with <<package phil_interface/
        package DiningPhilosopher/package i_Fork>>exported_i_Fork;
    out with <<package phil_interface/
        package DiningPhilosopher/package i_Fork>>imported_i_Fork;

  /* defines the factory process */
  process type o_Fork_factory;
    gate factory
      in with <<package phil_interface/package DiningPhilosopher/
       package o_Fork/package factory>>o_Fork_factory; /* ; zuviel  */
    channel nodelay
      from this via factory to env;
    endchannel;

    /* stores component keys here */
    dcl keys ComponentKeysSeq;

    /* creates a CO and returns a reference to it */
    exported as <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>generic_create
    procedure generic_create -> ComponentBase;
   dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
        task keys := Modify(keys, key, offspring);
      return offspring;
    endprocedure;

    /* creates a CO and returns a reference to it */
    exported as <<package phil_interface/package DiningPhilosopher/
        package o_Fork/package factory>>create_o_Fork
    procedure create_o_Fork -> o_Fork;
```

```
      dcl key ComponentKey;
      start;
    create co_instance;
      task key := call get_key to offspring;
      task keys := Modify(keys, key, offspring);
    return offspring;
  endprocedure;

  /* returns name of CO type */
  exported as <<package phil_interface/package DiningPhilosopher/
     package o_Fork/package factory>>get_co_type
  procedure get_co_type -> string;
    start;
    return 'DiningPhilosophers.o_Fork';
  endprocedure;

  /* returns reference to a CO */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>resolve_co
   procedure resolve_co(in key ComponentKey) -> ComponentBase
        raise InvalidIndex;
      dcl returnValue Pid;
      dcl i Integer := 1;
      start;
      task returnValue := Extract(keys,key);
      return returnValue;
   endprocedure;

  /* returns a list of CO keys */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>list_co
   procedure list_co -> ComponentKeySeq;
      start;
      return keys;
   endprocedure;

endprocess type o_Fork_factory;
/* \-------------------------------/ */

/* defines the CO type itself */
process type o_Fork;

  /* gates used for comm. with environment */
  gate initial
    in with  (<<package phil_interface/package DiningPhilosopher
          /package o_Fork>>o_Fork);
  gate provides
    in with <<package phil_interface/package DiningPhilosopher
          /package i_Fork>>exported_i_Fork; /* NOS */
    out with <<package phil_interface/package DiningPhilosopher
          /package i_Fork>>imported_i_Fork;

  /* /-------------------------------\ */
  /*     encapsules state variables     */
  process data_access(1,1);

    dcl reference_interaction_i_Fork exported_i_Fork;
    dcl reference_port_fork exported_i_Fork;
    dcl the_key string;


  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_port_fork
  procedure get_port_fork -> exported_i_Fork;
```

```
    start;
    return reference_port_fork;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_interaction_i_Fork
  procedure get_interaction_i_fork -> exported_i_Fork;
    start;
    return reference_interaction_i_Fork;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>set_port_fork
  procedure set_port_fork(in ref exported_i_Fork);
    start;
    reference_port_fork := ref;
  endprocedure;
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>set_interaction_i_Fork
  procedure set_interaction_i_fork(in ref exported_i_Fork);
    start;
    reference_interaction_i_Fork := ref;
  endprocedure;

endprocess data_access;
/* \--------------------------------/ */

/* channel artefact <--> state_access */
channel nodelay
  from artefact_a_ForkImpl to data_accessor
    with internal_state;
endchannel;


/* in case of CO type inheritance an instance of this type */
/* is contained in an "eODL-inherited" type. The following */
/* gate is used to interface with the state_access process */
/* of the inherited type. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;


/* /--------------------------------\ */
/*     manages artefact instances     */
process artefactmanagement(1,1);
  /* signals for delegation of artefact creations */
  signal create_a_ForkImpl_req;
  signal create_a_ForkImpl_res(Pid);
  /* Artefact-Pool and pointer in pool */
  dcl a_ForkImpl_seq PIdSeq := (. .);
  dcl a_ForkImpl_ptr Integer := 0;
  /* returns an artefact instance reference */
  /* implements a pool of size POOLSIZE */
  /* creates a new artefact if pool is not yet of */
  /* size POOLSIZE, otherwise returns the "next" */
  /* artefact. */
  exported procedure get_artefact_a_ForkImpl -> PId;
    dcl new_pid PId;
    start;
     decision length(a_ForkImpl_seq);
      (0:1):
        /* delegate creation of artefact */
```

```
          output create_a_ForkImpl;
            nextstate wait4response;
        else:
          task a_ForkImpl_ptr := a_ForkImpl_ptr+1;
          task { if (a_ForkImpl_ptr>length(a_ForkImpl_seq))
                 a_ForkImpl_ptr := 1;
               };
          return extract(a_ForkImpl_seq, a_ForkImpl_ptr);
      enddecision;
      /* delegation continued ... */
      state wait4response;
      input create_a_ForkImpl_res(new_pid);
      task a_ForkImpl_seq := a_ForkImpl_seq // new_pid;
      return offspring;
    endprocedure;
    start;
    nextstate wait_for_signal;
    /* create artefact instance for procedure */
    state wait_for_signal;
    input create_a_ForkImpl_req;
    create artefact_a_ForkImpl;
     output create_a_ForkImpl_res(offspring) to sender;
    nextstate -;
  endprocess;
  /* \--------------------------------/ */

  /* this is the interactionmanagementrepresentation */
  /* for interface i_fork. handles procedure calls   */
  /* from the environment */
  process interaction_i_Fork(0,1);
    exported as <<interface exported_i_Fork>>obtain_Fork
    procedure obtain_fork(in eater o_Philosopher)
            raise ForkNotAvailable;
      dcl p PId;
      start;
      task p := get_artefact_a_ForkImpl;
      call obtain_Fork(who, sender) to p;
      return;
        exceptionhandler defhandler;
         handle ForkNotAvailable;
         raise ForkNotAvailable;
      endexceptionhandler defhandler;
    endprocedure;
    exported as <<interface exported_i_Fork>>release_Fork
    procedure release_fork(in eater o_Philosopher)
      raise NotTheEater;
      dcl p PId;
      start;
      task p := get_artefact_a_ForkImpl;
      call obtain_Fork(who,sender) to p;
      return;
        exceptionhandler defhandler;
         handle NotTheEater;
         raise NotTheEater;
      endexceptionhandler defhandler;
    endprocedure;
  endprocess;

  /* portmanagement */
  exported as <<package phil_interface/
       package DiningPhilosopher>>provide_fork
  procedure provide_fork -> exported_i_Fork;
    start;
    return call provide_fork;
```

```
endprocedure;
exported as <<package philo_interface/
   package DiningPhilosopher/package o_Fork>>provide
procedure provide(s string) -> PId
     raise NoSuchPort;
  start;
  decision s;
     (='fork'): return call provide_fork;
    else: raise NoSuchPort;
  enddecision;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Fork>>port_connect
procedure port_connect(s string) -> PId
   raise NoSuchPort,AlreadyConnected;
  start;
  raise NoSuchPort;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Fork>>port_disconnect
procedure port_disconnect(s string) -> PId
    raise NoSuchPort,NotConnected;
  start;
  raise NoSuchPort;
endprocedure;
  /* get/set for internal variables */
  exported as <<package philo_definition/package DiningPhilosopher/
          package o_Fork_data>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;


/* computes key and instantiates        */
/* interactionmanagementrepresentations */
start;
task the_key := <<package eODL>>generate_key;
create interaction_i_Fork;
call set_interaction_i_Fork(offspring);
task reference_port_fork := offspring;
nextstate initial_state;

/* process instance set of artefact a_ForkImpl */
process  artefact_a_ForkImpl(0,2): a_ForkImpl;

/* connects artefact and imr */
channel interaction_i_fork_a_fork_impl nodelay
   from interaction_i_Fork to artefact_a_ForkImpl
      with procedure <<package phil_definition/package DiningPhilosophers
          /package i_Fork>>obtain_fork,
          procedure <<package phil_definition/package DiningPhilosophers
          /package i_Fork>>release_fork;
endchannel;

channel ch_i_Fork nodelay
   from env via provides to interaction_i_Fork
      with exported_i_Fork;
   from interaction_i_Fork to env via provides
      with imported_i_Fork;
endchannel;

 channel ch_initial nodelay
  from env via initial to this
```

```
        with config_o_Fork;
    endchannel;

  endprocess type o_Fork;
  /* \--------------------------------/ */

  /* instance sets for factory and co type */
  process factory(1,1): o_Fork_factory;
  process cos(0,): o_Fork;

  channel factory_to_co nodelay
    from factory to co via initial;
  endchannel;
  channel initial_to_env nodelay
    from env via initial to co via initial;
  endchannel;
  channel provides_to_env nodelay
    from cos via provides to env via provides;
    from env via provides to cos via provides;
  endchannel;
  channel uses_to_env nodelay
    from cos via uses to env via uses;
    from env via uses to cos via uses;
  endchannel;


endblock type o_Fork_CO;
/* \--------------------------------/ */

package i_Philosopher;

    interface exported_i_Philosoper;
      procedure set_name(in string, in Pid);
    endinterface;

    interface imported_i_Philosoper;
    endinterface;

  endpackage i_Philosopher;

  package o_Philosopher_data;

    interface internal_data;
      procedure get_port_observer -> i_Observer;
      procedure get_port_left -> i_Fork;
      procedure get_port_right -> i_Fork;
      procedure get_interaction_i_Fork -> imported_i_Fork;
      procedure get_interaction_i_Observer -> imported_i_Observer;
      procedure get_interaction_i_Philosopher -> exported_i_Philosopher;
      procedure set_port_observer(i_Observer);
      procedure set_port_left(i_Fork);
      procedure set_port_right(i_Fork);
      procedure set_interaction_i_Fork(imported_i_Fork);
      procedure set_interaction_i_Observer(imported_i_Observer);
      procedure set_interaction_i_Philosopher(exported_i_Philosopher);
    endinterface;

  endpackage;

  signallist a_PhilosopherImpl :=
    procedure <<package phil_definition/package DiningPhilosophers/
                    package i_Philosopher>>set_name;
```

```
process type a_PhilosopherImpl with
   use (a_PhilosopherImpl_in);;
  referenced;


block type o_Philosopher_CO;

  gate factory
    in with <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>o_Philosopher_factory;
  gate initial
    in with  <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher>>o_Philosopher;
  gate provides
    in with <<package phil_interface/package DiningPhilosophers/
            package i_Philosopher>>exported_i_Philosopher;
    out with <<package phil_interface/package DiningPhilosophers/
            package i_Philosopher>>imported_i_Philosopher;
  gate uses
    out with <<package phil_interface/package DiningPhilosophers/
            package i_Fork>>exported_i_Fork,
             <<package phil_interface/package DiningPhilosophers
            /package i_Observer>>exported_i_Observer;

  process type o_Philosopher_factory;
    gate factory
      in with <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>o_Philosopher_factory;
    channel nodelay
      from this via factory to env;
     endchannel;

     dcl keys ComponentKeysSeq;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>generic_create
     procedure generic_create -> ComponentBase;
      dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
       task keys := keys // key;
      return offspring;
    endprocedure;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>create_o_Philosopher
     procedure create_o_Philosopher -> o_Philosopher;
       dcl key ComponentKey;
       start;
      create co_instance;
       task key := call get_key to offspring;
       task keys := keys // key;
      return offspring;
    endprocedure;

    exported as <<package phil_interface/package DiningPhilosophers/
            package o_Philosopher/package factory>>get_co_type
    procedure get_co_type -> string;
      start;
      return 'o_Philosopher';
    endprocedure;

    /* returns reference to a CO */
```

```
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>resolve_co
   procedure resolve_co(in key ComponentKey) -> ComponentBase
      raise InvalidIndex;
      dcl returnValue Pid;
      dcl i Integer := 1;
      start;
      task returnValue := Extract(keys,key);
      return returnValue;
   endprocedure;

  /* returns a list of CO keys */
   exported as <<package phil_interface/package DiningPhilosophers/
          package o_Fork/package factory>>list_co
   procedure list_co -> ComponentKeySeq;
      start;
      return keys;
   endprocedure;

endprocess type;


process type o_Philosopher;
  gate initial
    in with  <<package phil_interface/package DiningPhilosophers/
          package o_Philosopher>>o_Philosopher;

  channel ch_initial nodelay
    from env via initial to this with o_Philosopher;
  endchannel;

  gate provides
    in with <<package phil_interface/package DiningPhilosophers/
          package i_Philosopher>>exported_i_Philosopher;
    out with <<package phil_interface/package DiningPhilosophers/
          package i_Philosopher>>imported_i_Philosopher;
  channel ch_provides nodelay
    from env via provides to interaction_i_Philosopher
          with exported_i_Philosopher;
    from interaction_i_Philosopher via provides to env
          with imported_i_Philosopher;
  endchannel;

  gate uses
    out with <<package phil_interface/package DiningPhilosophers/
          package i_Fork>>exported_i_Fork,
              <<package phil_interface/package DiningPhilosophers/
          package i_Observer>>exported_i_Observer;

  dcl the_key string;

  process data_access(1,1);
   dcl reference_interaction_i_Philosopher exported_i_Philosopher;
   dcl reference_interaction_i_Fork imported_i_Fork;
   dcl reference_interaction_i_Observer imported_i_Observer;
   dcl reference_port_observer exported_i_Observer := Null;
   dcl reference_port_left exported_i_Fork := Null;
   dcl reference_port_right exported_i_Fork := Null;

 exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_port_observer
    procedure get_port_observer -> i_Observer;
        start;
        return reference_port_observer;
```

```
     endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_port_observer
    procedure set_port_observer(ref i_Observer);
  start;
  task reference_port_observer := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_port_left
    procedure get_port_left -> i_Fork;
  start;
  return reference_port_left;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_port_left
    procedure set_port_left(ref i_Fork);
  start;
  task reference_port_left := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_port_right
    procedure get_port_right -> i_Fork;
  start;
  return reference_port_right;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_port_right
    procedure set_port_right(ref i_Fork);
  start;
  task reference_port_right := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>get_interaction_i_Fork
procedure get_interaction_req_i_Fork -> imported_i_Fork;
  start;
  return reference_interaction_i_Fork;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
          package o_Philosopher_data>>set_interaction_i_Fork
procedure set_interaction_i_Fork(ref imported_i_Fork);
  start;
  task reference_interaction_i_Fork := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_interaction_i_Observer
    procedure get_interaction_i_Observer -> imported_i_Observer;
  start;
  return reference_interaction_i_Observer;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_interaction_i_Observer
    procedure set_interaction_i_Observer(ref imported_i_Observer);
  start;
  task reference_interaction_i_Observer := ref;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>get_interaction_i_Philosopher
    procedure get_interaction_i_Philosopher -> exported_i_Philosopher;
  start;
  return reference_interaction_i_Philosopher;
endprocedure;
exported as <<package philo_definition/package DiningPhilosophers/
     package o_Philosopher_data>>set_interaction_i_Philosopher
    procedure set_interaction_i_Philosopher(ref exported_i_Philosopher);
```

```
   start;
   task reference_interaction_i_Philosopher := ref;
 endprocedure;

  endprocess;


/* channel artefact <--> state_access */
channel nodelay
  from artefact_a_ForkImpl to data_accessor
    with internal_state;
endchannel;


/* in case of CO type inheritance an instance of this type */
/* is contained in an "eODL-inherited" type. The following */
/* gate is used to interface with the state_access process */
/* of the inherited type. */
gate data_accessor
  in with internal_data;
channel route_state_access nodelay
  from
    env via data_accessor to data_access;
endchannel;

  process artefactmanagement(1,1);
    signal create_a_PhilosopherImpl_req;
    signal create_a_PhilosopherImpl_res(Pid);
    dcl a_PhilosopherImpl_ptr PId := Null;
    exported procedure get_artefact_a_PhilosopherImpl;
      dcl new_pid PId;
      start;
       decision a_PhilosopherImpl_ptr;
        (Null):
           output create_a_PhilosopherImpl;
            nextstate wait4response;
        else:
          return a_PhilosopherImpl_ptr;
      enddecision;

      state wait4response;
       input create_a_PhilosopherImpl_res(new_pid);
      task a_PhilosopherImpl_ptr := new_pid;
      return offspring;
    endprocedure;
    start;
    nextstate wait_for_signal;
    state wait_for_signal;
    input create_a_PhilosopherImpl;
    create artefact_a_PhilosopherImpl;
    nextstate -;
  endprocess;

  channel nodelay
  from
    artefact_a_PhilosopherImpl
    to state_accessor
    with internal_state;
  endchannel;

  process interaction_i_Philosopher(0,1);
     exported as <<package phil_definition/package DiningPhilosophers/
          package i_Philosopher>>set_name
    procedure set_name(in name string);
```

```
    dcl p PId;
    start;
     task p := get_artefact_a_PhilosopherImpl;
     call set_name(name,sender) to p;
     return;
   endprocedure;
 endprocess;

process interaction_i_Fork(0,1);
   exported as <<package phil_definition/package DiningPhilosophers/
         package i_Fork>>release_fork
  procedure obtain_fork(in eater o_Philosopher, in server PId);
    start;
     call obtain_fork(eater) to server;
     return;
   endprocedure;

  exported as <<package phil_definition/package DiningPhilosophers/
    package i_Fork>>release_fork
  procedure release_fork(in eater o_Philosopher, in server PId);
    start;
     call obtain_fork(eater) to server;
     return;
   endprocedure;
 endprocess;

 process interaction_i_Observer(0,1);
    dcl carry_PhilosopherState PhilosopherState;
    dcl consumer PId;
    start;
    nextstate signal_handler;
    state signal_handler;
    input PhilosopherState(carry_PhilosopherState, consumer);
    output PhilosopherState(carry_PhilosopherState) to consumer;
    nextstate -;
 endprocess;


process artefact_a_PhilosopherImpl(0,1): a_PhilosopherImpl;

  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_observer
  procedure link_observer -> exported_i_Observer;
     start;
      decision call get_port_observer;
     (Null): call set_port_observer(ref);
      else: raise AlreadyConnected;
      enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_observer
  procedure unlink_observer -> exported_i_Observer;
     start;
      decision call get_port_observer;
     (Null): raise NotConnected;
      else: call set_port_observer(ref);
      enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_left
  procedure link_left(ref imported_i_Fork)
        raise AlreadyConnected;
     start;
      decision call get_port_left;
```

```
    (Null): call set_port_left(ref);
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_left
  procedure unlink_left
        raise NotConnected;
    start;
     decision call get_port_left;
    (Null): raise NotConnected;
     else: call set_port_left(Null);
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_right
  procedure link_right(ref imported_i_Fork)
        raise AlreadyConnected;
    start;
     decision call get_port_right;
    (Null): call set_port_right ( ref );
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_right
  procedure unlink_right
        raise NotConnected;
    start;
     decision call get_port_right;
    (Null): raise NotConnected;
     else: call set_port_right(Null);
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>link_observer
  procedure link_observer(ref imported_i_Observer)
        raise AlreadyConnected;
    start;
     decision call get_port_observer;
    (Null): call set_port_observer(ref);
     else: raise AlreadyConnected;
     enddecision;
  endprocedure;
  exported as <<package philo_interface/package DiningPhilosophers/
        package o_Philosopher>>unlink_observer
  procedure unlink_observer
        raise NotConnected;
    start;
     decision call get_port_observer;
    (Null): raise NotConnected;
     else: call set_port_observer(Null);
     enddecision;
  endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Philosopher>>provide
procedure provide(s string) -> PId
     raise NoSuchPort;
  start;
   enddecision;
endprocedure;
exported as <<package philo_interface/
     package DiningPhilosopher/package o_Philosopher>>port_connect
procedure link(ref Pid, s string)
```

```
      raise NoSuchPort,AlreadyConnected;
    start;
    decision s;
      (='observer'): call link_observer(ref);
      (='left'): return call link_left(ref);
      (='right'): return call link_right(ref);
      else: raise NoSuchPort;
    return ;
  endprocedure;
  exported as <<package philo_interface/
       package DiningPhilosopher/package o_Philosopher>>port_disconnect
  procedure unlink(s string) -> PId
      raise NoSuchPort,NotConnected;
    start;
    decision s;
      (='observer'): call unlink_observer;
      (='left'): return call unlink_left;
      (='right'): return call unlink_right;
      else: raise NoSuchPort;
    return ;
  endprocedure;

 start;
    task the_key := <<package eODL>>generate_key;
 create interaction_i_Fork;
    call set_interaction_i_Fork(offspring);
 create interaction_i_Philosopher;
 call set__interaction_i_Philosopher(offspring);
 create interaction_i_Observer;
 call set__interaction_i_Observer(offspring);
 nextstate initial_state;

  exported as <<package phil_interface/package DiningPhilosophers/
       package o_Philosophers>>get_key
  procedure get_key -> ComponentKey;
    start;
    return the_key;
  endprocedure;

endprocess type;

process factory(1,1): o_Fork_factory;
process cos(0,): o_Fork;


channel factory_to_co nodelay
  from factory to co via initial;
endchannel;

channel initial_to_env nodelay
  from co via initial to env via initial;
endchannel;

channel provides_to_env nodelay
  from co via provides to env via provides;
endchannel;

channel uses_to_env nodelay
  from co via uses to env via uses;
endchannel;

channel factory_to_env nodelay
  from env via factory to  factory;
endchannel;
```

```
  endblock type;

package o_Observer_data;
    interface internal_data;
      procedure get_interaction_i_Observer -> exported_i_Observer;
      procedure get_port_observer -> exported_i_Observer;
      procedure set_interaction_i_Observer(exported_i_Observer);
      procedure set_port_observer(exported_i_Observer);
    endinterface;
  endpackage;

  signal PhilosopherState(PState, Pid);

  signallist a_Observer_in =
       <<package phil_definition/package DiningPhilosophers>>PhilosopherState;

  /* artefact: referenced definition */
  process type a_ForkImpl with
     use (a_Observer_in);;
    referenced;



  block type o_Observer_CO;


    gate factory
      in with <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>o_Observer_factory;
    gate initial
      in with  <<package phil_interface/package DiningPhilosophers/
              package o_Observer>>o_Observer;
    gate provides
      in with <<package phil_interface/package DiningPhilosophers/
              package i_Observer>>exported_i_Observer;
      out with <<package phil_interface/package DiningPhilosophers/
              package i_Observer>>imported_i_Observer;

    process type o_Observer_factory;
      gate factory
        in with <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>o_Observer_factory;
      channel nodelay
        from this via factory to env;
       endchannel;

       dcl keys ComponentKeysSeq;

      exported as <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>generic_create
       procedure generic_create -> ComponentBase;
        dcl key ComponentKey;
         start;
        create co_instance;
         task key := call get_key to offspring;
         task keys := keys // key;
        return offspring;
      endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
              package o_Observer/package factory>>create_o_Observer
       procedure create_o_Observer -> o_Observer;
```

```
        dcl key ComponentKey;
        start;
       create co_instance;
        task key := call get_key to offspring;
        task keys := keys // key;
       return offspring;
     endprocedure;

     exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>get_co_type
     procedure get_co_type -> string;
       start;
       return 'o_Observer';
     endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>resolve_co
      procedure resolve_co(in key ComponentKey) raise InvalidIndex;
        dcl returnValue Pid;
        dcl i Integer := 1;
        start;
        task returnValue := Extract(keys,key);
        return returnValue;
      endprocedure;

      exported as <<package phil_interface/package DiningPhilosophers/
             package o_Observer/package factory>>list_co
      procedure list_co -> ComponentKeySeq;
        start;
        return keys;
      endprocedure;

   endprocess type;

   process type o_Observer;
     gate initial
       in with  <<package phil_interface/
          package DiningPhilosopher/package o_Observer>>o_Observer;
     gate provides
       in with <<package phil_interface/
          package DiningPhilosopher/package o_Observer>>exported_i_Observer;
       out with <<package phil_interface/package DiningPhilosopher/
          package o_Observer>>imported_i_Observer;


   dcl reference_interaction_i_Observer exported_i_Fork;
   dcl reference_port_observer exported_i_Fork;
   dcl the_key string;

   process data_access(1,1);
     exported as <<package philo_definition/
        package DiningPhilosopher/package o_Fork_data>>get_port_observer
     procedure get_port_fork -> exported_i_Observer;
       start;
       return reference_port_observer;
     endprocedure;
     exported as <<package philo_definition/
        package DiningPhilosopher/package
o_Fork_state>>get_interaction_i_Observer
     procedure get_interaction_i_Observer -> exported_i_Observer;
       start;
       return reference_interaction_i_Observer;
     endprocedure;
   endprocess;
```

```
      exported as <<package philo_definition/
         package DiningPhilosopher/package o_Fork_data>>set_port_observer
      procedure set_port_fork(in ref exported_i_Observer);
        start;
        reference_port_observer := ref;
      endprocedure;
      exported as <<package philo_definition/
         package DiningPhilosopher/package
o_Fork_state>>set_interaction_i_Observer
      procedure set_interaction_i_Observer(exported_i_Observer);
        start;
        reference_interaction_i_Observer := ref;
      endprocedure;
    endprocess;

    gate state_accessor
      in with internal_state;
    channel route_state_access nodelay
      from
        state_accessor via state_accessor to env;
    endchannel;
    channel nodelay
      from artefact_a_Observer to state_accessor
        with internal_state;
    endchannel;


    process artefactmanagement(1,1);
      signal create_a_Observer_req;
      signal create_a_Observer_res(Pid);
      dcl a_Observer Pid := Null;
      exported procedure get_artefact_a_Observer;
        dcl new_pid PId;
        start;
         decision a_ForkImpl;
          (Null):
             output create_a_ForkImpl;
              nextstate wait4response;
          else:
            return a_Observer;
        enddecision;

        state wait4response;
        input create_a_ForkImpl_res(new_pid);
        task a_ForkImpl_seq := a_ForkImpl_seq // new_pid;
        return offspring;
      endprocedure;
      start;
      nextstate wait_for_signal;
      state wait_for_signal;
      input create_a_Observerl;
      create artefact_a_Observer;
      nextstate -;
    endprocess;

    process interaction_i_Observer(0,1);
      dcl p PId;
       dcl e_PState pstate;
      start;
      nextstate wait4signal;
       state wait4signal;
      input PhilosopherState(pstate);
      task p := get_artefact_a_Observer;
      output PhilosopherState(pstate, sender) to p;
```

```
    nextstate -;
  endprocess;

process  artefact_a_Observer(0,1): a_Observer;

    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>provide_observer
    procedure provide_observer -> exported_i_Observer;
      start;
      return call get_provide_observer to data_access;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>provide
    procedure provide(s string) -> PId
         raise NoSuchPort;
      start;
      decision s;
       ('observer'): return reference_port_observer;
       else: raise NoSuchPort;
       enddecision;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>port_connect
    procedure port_connect(s string) -> PId
        raise NoSuchPort,AlreadyConnected;
      start;
      raise NoSuchPort;
    endprocedure;
    exported as <<package philo_interface/
         package DiningPhilosopher/package o_Observer>>port_disconnect
    procedure port_disconnect(s string) -> PId
        raise NoSuchPort,NotConnected;
      start;
      raise NoSuchPort;
    endprocedure;


  start;
  task the_key := <<package eODL>>generate_key;
  create interaction_i_Observer;
  call set_interaction_i_Observer(offspring) to data_access;
  call set_port_observer(offspring) to data_access;
  nextstate initial_state;

  channel ch_provides nodelay
     from env via provides to interaction_i_Observer
        with exported_i_Observer;
     from interaction_i_Fork to env via provides
        with imported_i_Observer;
  endchannel;

   channel ch_initial_port nodelay
     from env via initial to portmanagement
        with config_o_Observer;
  endchannel;

   channel ch_initial nodelay
     from env via initial to this with imported_o_Observer;
     from this via initial to env with exported_o_Observer;
   endchannel;

  exported as <<package phil_interface/package DiningPhilosophers/
       package o_Observer>>get_key
  procedure get_key -> ComponentKey;
```

```
      start;
      return the_key;
    endprocedure;
  endprocess type;

  process factory(1,1): o_Observer_factory;
  process co(0,): o_Observer;

  channel factory_to_co nodelay
    from factory to co via initial;
  endchannel;

  channel initial_to_env nodelay
    from co via initial to env via initial;
  endchannel;

  channel provides_to_env nodelay
    from co via provides to env via provides;
  endchannel;

  channel uses_to_env nodelay
    from co via uses to env via uses;
  endchannel;


  endblock type;

endpackage;

endpackage DiningPhilosophers;

endpackage phil_definition;
```

Appendix II

Information processing and tool support

II.1 Introduction

The existence of a simple and complete **metamodel** provides a stable basis for software development, even in complex application areas. In order to make the technique usable and in particular to provide ease of use to developers, appropriate tool support is needed. In general, such tools may support the modelling process itself, like an editor or/and simulator, or they may cover more phases, like implementation and **deployment** on **target platforms**.

Due to the widespread variety of tasks to be supported by tools processing a single model from beginning of specification to **deployment** to instantiation, it is expected that different tools will be used in a tool chain. Thus, the issue of having an interchange format from one tool to another arises. For this purpose at least one standardized notation exists by default. Due to the appliance of OMG MOF, a XML-based representation is implied for the **metamodel** and thus can be used as an interchange format between different tools.

The concrete definition of tools and their functionality cannot be subject to standardization. Although, in practice, single tools may be arbitrarily designed or actually consist of tool chains, the subsequent clauses subdivide tool issues that focus on certain aspects. Actual tools may span several of those aspects.

II.2 Modelling tool issues

Tools dealing with manipulation of model information may use an arbitrary representation of the model with the only restriction that each representation has to have an appropriate **metamodel** mapping. Such representations may range from programmatic languages to graphical notations, like UML. Any tool supporting the processing of such a representation can be used and no restrictions are made.

Since the **metamodel** covers almost the entire life cycle of software, there is a variety of possible modelling issues, each certainly done with tool support. A model may be stepwise enlarged by adding additional information in several iterations of modelling at different points of time. Thus, a collection of already existing **CO types** may be used to specify an **assembly** later on, and a concrete assignment of this **assembly** to a **target platform** may be given thereafter. In general, the modelling issues are the following, ordered by time of application:

- **CO type** specification;
- **assembly** specification;
- implementation packaging (of **software components**);
- environment modelling;
- assignment of **software components** on a **target platform**;
- instantiation of **CO types**' respective **assemblies**.

The specification of **CO types** according to the **metamodel** is the first step to take. After having such types, assemblies may be defined. Each type may be used in an arbitrary number of assemblies. The next step is to provide the implementation for the whole **assembly** containing the implementation code for used **CO types** grouped in **software components**. The packaging of implementations may be done by archive tools, like zip. After having all these provided, the model in combination with the **assembly** implementation package can be shipped to be deployed at customers' platforms. In order to deploy an **assembly**, the distribution of **COs** has first to be determined. During this process, the model is enriched by additional information, which is mainly related to the concrete environment and the special business case of a customer. The model of the target environment and the model of the **assembly** are compared to find a proper assignment for each **CO** to a node of the platform. This can be done manually, preferably by the system administrator or semi-automatically by an automatic function providing a solution for the **initial configuration** of the **assembly**. In either case, the requirements of each **CO** of the **assembly** on the target environment have to be fulfilled. Where to get the environment model is not specified. It could be obtained directly from the target environment, in which case a special architecture would be required. As a result of the assignment step, the model contains information about where to install which **software component**. The actual instantiation of **CO types** or assemblies may be part of the model. An appropriate tool would have to keep the model up to date during **runtime**.

II.3 Generator tool issues

Tools providing support for the implementation of model entities may save a lot of efforts compared to doing implementations manually. Code generation may be done for all implementation languages which have a mapping from the **metamodel**. In general, the following information may be generated from the model:

- skeletons for **CO types**;
- code for Quality of Service specifications.

The generated code may offer a framework for the implementer to serve as a basis for **CO type** implementations. As long as no behavioural aspects are contained in the **metamodel**, no business logic implementation may be generated automatically. Instead it has to be inserted by hand.

II.4 Deployment tool issues

Deployment including instantiation of assemblies on a customer's **target platform** requires support by proper tools, but also depends on appropriate support by the platform itself. Therefore, actual tools for **deployment** are tightly coupled to concrete platform architectures that they have to interact with. Tools cannot be independent as long as there is no standardized platform architecture which they can collaborate with.

In general terms, the process of **deployment** comprises several tasks beginning from the determination of a proper distribution, to the installation and instantiation of software. The common **deployment** tasks may be handled by tools:

- environment modelling;
- assignment of **software components** on a **target platform**;
- installation of **software components** on a **target platform**;
- **assembly** respective **CO type** instantiation;
- constraint and action processing.

The tasks of environment modelling and of assignment of **software components** to a **target platform** were already mentioned in the context of modelling tool issues. In fact, during these tasks, models are extended, which is the reason why they were handled there. Actually, these tasks are in most cases expected to be performed as part of the **deployment** of an **assembly**. As already mentioned, a platform may support tools to gain environment modelling information. Having done these tasks and determined a proper assignment of **software components** to a **target platform**, the next step is to upload and install the software on the specified node. After this, the **assembly** or **CO types** may be instantiated with the help of another tool or platform capability. Lastly, at **runtime**, constraints and actions contained in the model have to be processed by some means.

In Figure II.1, the chain of tool issues is depicted from modelling to **deployment**.

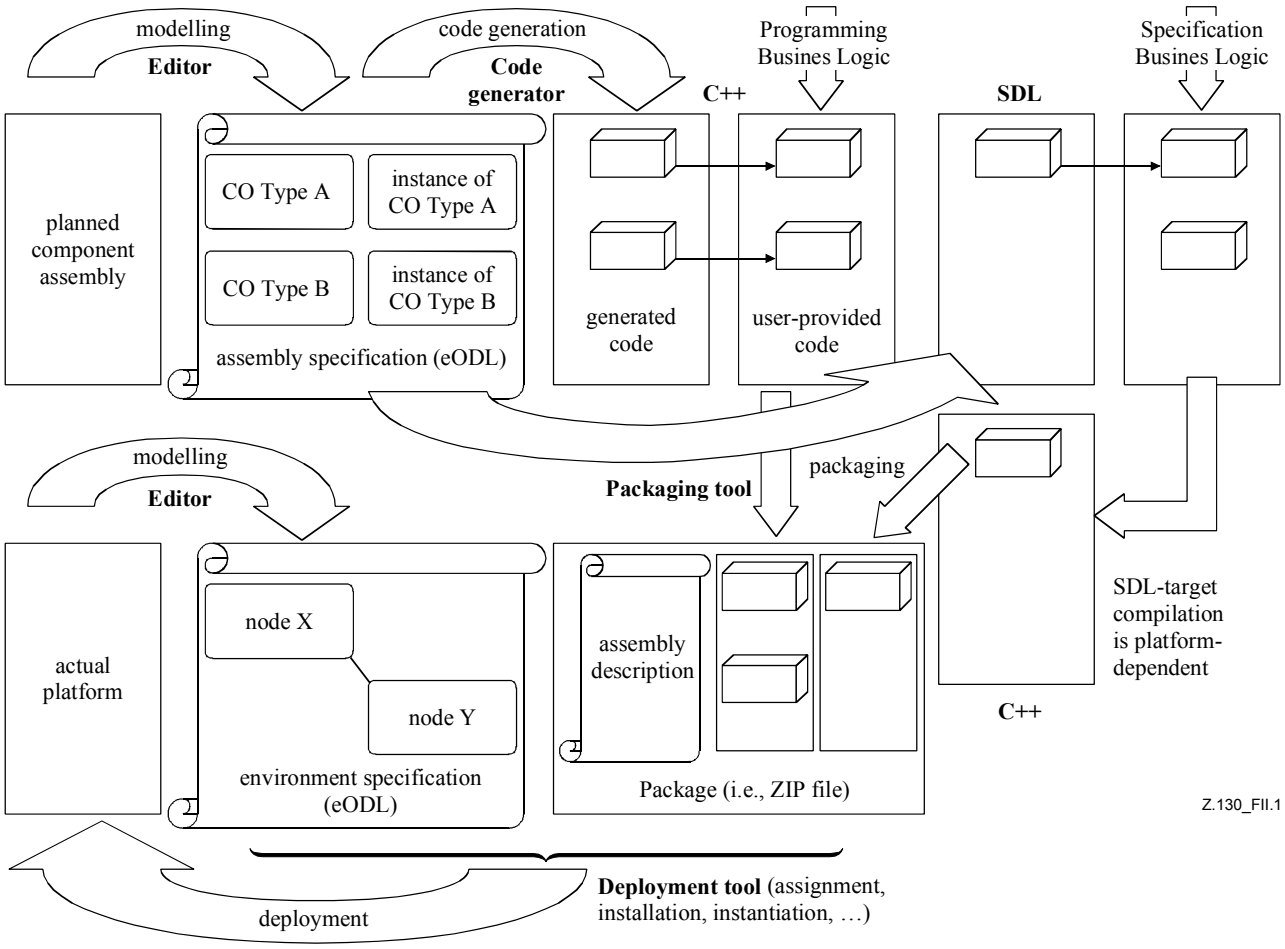


Figure II.1/Z.130 – Information processing

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2004