INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.130

(02/99)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT)

ITU object definition language

ITU-T Recommendation Z.130

(Previously CCITT Recommendation)

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	Z.120–Z.129
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.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

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

ITU-T RECOMMENDATION Z.130

ITU OBJECT DEFINITION LANGUAGE

Summary

This Recommendation specifies the ITU Object Definition Language (ITU-ODL). ITU-ODL is used for the specification of systems from the perspective of the Open Distributed Processing (ODP) computational viewpoint [3]. It defines templates for operational interfaces, stream interfaces, multiple-interface objects, and object groups.

ITU-ODL is an extension of the ODP Interface Definition Language (ODP-IDL, [8]) with additions to support the specification of ODP computational viewpoint concepts on a syntactic level. ITU-ODL is a superset of ODP-IDL. This relationship between ITU-ODL and ODP-IDL supports the construction of systems via OMG specified Object Request Broker (ORB) implementations [1]. The readers of this Recommendation are expected to be familiar with ODP-IDL.

Source

ITU-T Recommendation Z.130 was prepared by ITU-T Study Group 10 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on the 12th of February 1999.

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The 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 Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 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 term *recognized operating agency (ROA)* includes any individual, company, corporation or governmental organization that operates a public correspondence service. The terms *Administration, ROA* and *public correspondence* are defined in the *Constitution of the ITU (Geneva, 1992)*.

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The 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, the 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

		Page
1	Scope	1
2	References	2
3	Abbreviations	2
4	Definitions	3
5	Foundations and rules	3
5.1	Definitions and conventions	3
5.1.1	Definitions	3
5.1.2	Graphical conventions	4
5.2	Naming and scoping	5
5.3	Interface template, object template and object group template separation and sharing	7
5.3.1	Data types	7
5.3.2	Operations	7
5.3.3	Flows	8
5.3.4	Interface templates	8
5.3.5	Object templates	8
5.3.6	Scoping rules	9
5.4	Behaviour	9
5.5	Inheritance	10
5.5.1	Introduction and motivation	10
5.5.2	Definitions	10
5.5.3	Inheritance in construct declarations	11
6	Specification of ITU-ODL	17
6.1	Type and constant declaration	17
6.1.1	Structure	17
6.1.2	Example of type and constant declarations	17
6.2	Interface template	18
6.2.1	Structure	18
6.2.2	Interface template inheritance	18
6.2.3	Interface template behaviour specification	18
6.2.4	Operational interface signature	19
6.2.5	Operational interface attributes	19
6.2.6	Stream (flow) signature	20
6.2.7	Example of interface template declaration	21

Page

6.3 Object template 21
6.3.1 Structure 21
6.3.2 Object template inheritance 22
6.3.3 Object template behaviour specification 22
6.3.4 Required interface templates 22
6.3.5 Supported interface templates 22
6.3.6 Object template initialization specification 23
6.3.7 Example of object template declaration 23
6.4 Object group template 24
6.4.1 Structure 24
6.4.2 Object group template inheritance 25
6.4.3 Object group template predicate specification 25
6.4.4 Member object templates and group templates 25
6.4.5 Contracts 25
6.4.6 Example of group template declaration 26

Annex A – BNF 26
A.1 Compliance 26
A.2 Lexical conventions 26
A.3 Keywords 27
A.4 Extended BNF notation 27
A.5 Syntax 27
A.5.1 Module syntax 28
A.5.2 Group syntax 28
A.5.3 Object syntax 28
A.5.4 Interface syntax 29
A.5.5 (Operational) interface syntax 29
A.5.6 (Stream) interface syntax 29
A.5.7 Supporting definition syntax 30

Annex B – Mapping to SDL and ASN.1 32
B.1 Motivation 32
B.2 Basic requirements 32
B.3 Structure 33
B.4 Scoped names 33
B.5 Module mapping 33
B.6 Interface template, operation, flow and attribute mapping 34

Page

B.7 Interface template inheritance 37
B.8 Mapping for object templates 37
B.9 Mapping for object group templates 39
B.10 Mapping for constants 40
B.11 Mapping for basic data types 40
B.12 Mapping for constructed data types 41
B.12.1 Mapping for structure types 41
B.12.2 Mapping for union 41
B.12.3 Mapping for enumeration 42
B.12.4 Mapping for sequence types 42
B.12.5 Mapping for strings 42
B.12.6 Mapping for arrays 42
B.13 Mapping for exceptions 43
B.14 Additional definitions 43

Annex C – Mapping to C++ 43
C.1 Motivation 43
C.2 Basic requirements 44
C.3 Structure 44
C.4 Scoped names 44
C.5 Module mapping 44
C.6 Interface template, operation, flow and attribute mapping 45
C.6.1 Behaviour and usage clauses 45
C.6.2 Flows 45
C.6.3 Interface template inheritance 45
C.7 Mapping for object templates 45
C.7.1 Required interface specification 45
C.7.2 Supported interface specification 45
C.7.3 Initialization specification 46
C.7.4 Inheritance 46
C.7.5 Example 46
C.8 Mapping for group templates 48
C.9 Mapping for constants 48
C.10 Mapping for basic data types 48
C.11 Mapping for constructed data types 48
C.12 Mapping for exceptions 48

Page

Appendix I – Quality of service 49
I.1 Motivation 49
I.2 Syntax 49
I.3 Example 50
I.4 Mapping to SDL 50
Appendix II – Comparison of ITU-ODL with ODP-IDL and TINA-ODL 50
II.1 ITU-ODL objective vs. ODP-IDL objective 50
II.2 Object model 51
II.3 ITU-ODL syntax vs. ODP-IDL syntax 51
II.3.1 General syntax 51
II.3.2 Interface syntax 51
II.3.3 Operation syntax 51

Recommendation Z.130

ITU OBJECT DEFINITION LANGUAGE

(Geneva, 1999)

1 Scope

ITU-ODL has been developed to:

- document computational specifications. For example, a service component can be described from the computational viewpoint using an ITU-ODL specification;
- provide syntax suitable for developing software engineering support applications such as ITU-ODL parsers, code generators, computational specification editors and related CASE tools.

ITU-ODL is an extension of the ODP Interface Definition Language (ODP-IDL) [8]. ITU-ODL supports features that are not (currently) covered by ODP-IDL. These stem from the computational language of the Reference Model for Open Distributed Processing [3] and include multiple interface object templates, group templates, stream interface templates and Quality of Service (QoS) descriptions[1]. A complete comparison of ITU-ODL and ODP-IDL can be found in Appendix II.

It should be noted that ITU-ODL is strongly influenced by the work done in the TINA Consortium according to the language TINA-ODL [4]. However, some concepts could not be adopted and need to be changed. Especially, the group template concept has a new semantic.

ITU-ODL provides a syntax to describe static aspects of the computational viewpoint of ODP systems. This means that the language covers the structures and signatures of these systems but not the behaviour (in a formal manner). Aspects which can be expressed by using the ITU-ODL syntax include:

- the description of computational object templates which support multiple interface templates. The different supported interface templates represent logically distinct services provided by the same object;
- the ability of an object to handle multiple instances of the same interface template. This allows the object to maintain client specific contexts;
- the ability to control the access and visibility of parts of an object functionality;
- the possibility to describe the services/functionality an object needs from its environment. This feature allows the checking of the compatibility of interface templates on a static specification level;
- the ability of specification structuring using object group templates. Object templates which share a common property can be clustered in an object group template. Examples include implementation aspects as well as management issues;
- the possibility to describe static aspects of stream interface templates;
- the ability to associate QoS attributes to operations and flows (this concept is described in Appendix I).

ITU-ODL is a basis for describing reusable software components.

---

[1] Quality of Service notation is contained in Appendix I.

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; all 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.

[1] OMG Adopted Specification (1998), *The Common Object Request Broker: Architecture and Specification, Revision 2.2*.

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

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

[4] TINA-C Specification (1996), *Object Definition Language Manual, Version 2.3*.

[5] ITU-T Recommendation Z.100 (1993), *CCITT Specification and Description Language (SDL).*

[6] ITU-T Recommendation Z.100 (1993) Add.1 (1996), *CCITT Specification and Description Language (SDL) – Addendum 1*.

[7] ITU-T Recommendation Z.105 (1995), *SDL combined with ASN.1 (SDL/ASN.1).*

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

[9] http://www.fokus.gmd.de/research/cc/platin/products/y-sce/ – An acceptor for ITU-ODL.

[10] ISO/IEC 14882:1998, *Programming languages – C++*.

3 Abbreviations

This Recommendation uses the following abbreviations:

CASE	Computer-Aided Software Engineering
CORBA	Common Object Request Broker Architecture
DPE	Distributed Processing Environment
IDL	Interface Definition Language
ODP	Open Distributed Processing
ORB	Object Request Broker

4 Definitions

This Recommendation makes use of the following terms defined in Recommendations [2], [3] and [8].

X.902	X.903	X.920
behavioural compatibility client object environment of an object inheritance instantiation of a template object quality of service server object service template	announcement computational interface computational object flow (interaction) group interrogation operation operation interface signature stream signature interface termination	exception file module multiple inheritance

NOTE – There is a clash in the definition of the term "object" in ODP and in the OMG. Since the ITU-ODL language includes ODP-IDL as a subset, the term "object" may be used as well to refer the OMG definition. Wherever this is the case, it is specially noted.

5 Foundations and rules

This clause introduces the principles of ITU-ODL, its metastructure and semantics. In this clause ITU-ODL syntax is presented for illustration purposes. The meaning of this syntax is explained as it is introduced, but the reader is reminded that a detailed presentation of ITU-ODL syntax is the subject of the following clause.

Initially, basic definitions and conventions are presented. The rules for basic naming and scoping follow. Further naming and scoping rules are added along with each architectural addition. A feature of ITU-ODL is its ability to share and reuse existing specifications. Following this, the principles are presented upon which behaviour is specified. As stated above, one form of specification reuse is through composition, while a second form is through specialization or inheritance. The specific inheritance architecture adopted in ITU-ODL is presented in 5.5.

Throughout this clause, rules are highlighted by labels of the form Rn (e.g. R1, R2, etc). Definitions of terms are indicated with labels of the form Dn (e.g. D1, D2, etc).

5.1 Definitions and conventions

5.1.1 Definitions

The following definitions are used in the remainder of this Recommendation:

5.1.1.1 (D1) construct: An ITU-ODL construct is either an object group template, object template, interface template, operation, or flow. The signature specification of an ITU-ODL construct is declared in ITU-ODL.

5.1.1.2 (D2) supporting definition: Definitions of data types, constants, and exception declarations are called supporting definitions.

Additional definitions are introduced in the following subclauses, as additional concepts are encountered.

5.1.2 Graphical conventions

The following conventions are applied to the graphical representation of the examples given in the remainder of this Recommendation.

- Object templates are represented as boxes (rectangles) (see Figure 1).

Figure 1/Z.130 – Object templates

- A supported operational interface template is represented as a filled rectangle contiguous with the box representing the object template to which it belongs (see Figure 2). Some interface templates might be emphasized by the use of a special pattern. An operation is represented as an arrow pointing to the operational interface template to which it belongs. Other elements of operational interface templates are not represented.

Figure 2/Z.130 – Object templates with supported interfaces

- A supported stream interface template is represented as a rectangle containing open and/or filled circles, contiguous with the box representing the object template to which it belongs (see Figure 3). Some interface templates might be emphasized by the use of a special pattern.
- A flow sink is represented as an arrow pointing to a filled circle on the stream interface template to which it belongs. A flow source is represented as an arrow pointing away from an open circle on the stream interface template to which it belongs. Other elements of stream interface templates are not represented.

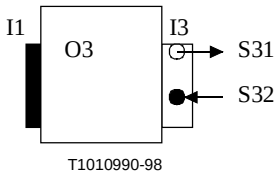

Figure 3/Z.130 – Object template with source and sink

- Object group templates are represented by dashed line boxes. Containment in an object group template is represented by containment in a dashed line box (see Figure 4).

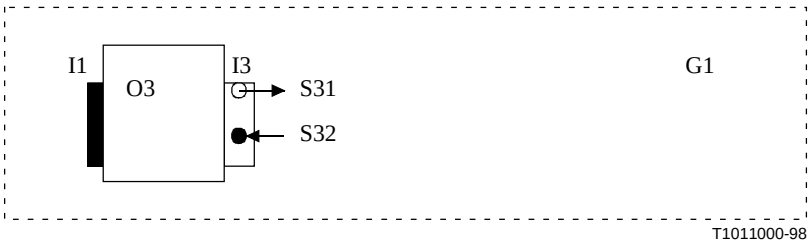

Figure 4/Z.130 – Group template

Having introduced the basic terms and graphical conventions used throughout this Recommendation, the naming and scoping framework of ITU-ODL will now be examined.

5.2 Naming and scoping

Naming and scoping rules are defined to enable the unambiguous identification of ITU-ODL constructs. For a comparative analysis of related work, see also comparable rules in ODP-IDL [8].

(R1) An entire ITU-ODL file forms a naming scope.

(R2) The following kinds of definitions form nested scopes:

- Module;
- Object group template;
- Object template;
- Interface template;
- Structure;
- Union;
- Operation; and
- Exception.

For example, the following ITU-ODL definitions are contained in one file. It specifies a module, M1, containing an object group template, G1, containing an object template, O1, containing an interface template, I1, containing a data type, DataType1, and an operation, operation1. M1 has global scope. G1 is scoped inside M1. O1 is scoped inside G1. I1 is scoped inside O1. DataType1 and operation1 are scoped inside I1.

```
module M1 {
        ...
        group G1 {
                ...
                CO O1 {
                        ...
                        interface I1 {
                                ...
                                typedef … DataType1;
                                ...
                                void operation1(in DataType1 variable11…);
                                ...
                        }; // end of I1
                }; // end of O1
        }; // end of G1
}; // end of M1
```

(R3) Identifiers for the following kind of definitions are scoped:

- Object group templates;
- Object templates;
- Interface templates;
- Operations;
- Data types;
- Constants;
- Enumeration values;
- Exceptions; and
- Attributes.

(R4) An identifier can only be defined once in a scope. Identifiers can be redefined in nested scopes.

(R5) Identifiers are case insensitive.

(R6) Identifiers defined in a scope are available for immediate use within that scope.

(R7) A qualified name (one of the form <scoped-name>::<identifier>) is resolved by locating the definition of <identifier> within the scope. The identifier must be defined directly in the scope. The identifier is not searched for in enclosing scopes.

For example, based on the ITU-ODL example above, the qualified name of G1 is M1::G1. Similarly, the qualified name of DataType1 is M1::G1::O1::I1::DataType1.

(R8) An unqualified name (one of the form <identifier>) can be used within a particular scope. It will be resolved by successively searching farther out in enclosing scopes. Once an unqualified name is used in a scope, it cannot be redefined.

For example, the ITU-ODL text below shows DataType1 defined in the scope of module M1. It is then used (as an unqualified name) within the scope of interface template I1. Further within the scope of I1, DataType1 is defined again. This second definition of DataType1 is illegal. If the second definition was placed before the operation1 statement, the redefinition would be legal, and this second definition would be used to resolve the unqualified name in the operation1 statement.

```
module M1 {
        ...
        typedef … DataType1;
        interface I1 {
                ...
                void operation1(in DataType1 variable11…);
                ...
                typedef … DataType1; // Illegal statement
                ...
        }; // end of I1
}; // end of M1
```

(R9) Every ITU-ODL definition in a file has a global name within that file. The rule to create a global name is the same as in ODP-IDL [8]:

"Prior to starting to scan a file containing an ODP-IDL specification, the name of the current root is initially empty ("") and the name of the current scope is initially empty (""). Whenever a `module` keyword is encountered, the string "::" and the associated identifier are appended to the name of the current root; upon detection of the termination of the `module`, the trailing "::" and identifier are deleted from the name of the current root. Whenever an

`interface`, `struct`, `union` or `exception` keyword is encountered, the string "::" and the associated identifier are appended to the name of the current scope; upon detection of the termination of the `interface`, `struct`, `union` or `exception`, the trailing "::" and identifier are deleted from the name of the current scope. Additionally, a new, unnamed, scope is entered when the parameters of an operation declaration are processed; this allows the parameter names to duplicate other identifiers; when parameter processing has completed, the unnamed scope is exited.

The global name of an ODP-IDL definition is the concatenation of the current root, the current scope, a "::", and the <identifier> which is the local name for that definition."

Additionally, for this purpose, object group template and object template are treated similarly to interface (template), struct, union, and exception.

5.2.1 (D3) tagged name: A tagged name is one of the form <scoped_name>"."<scoped_name>.

This construct is used to have a qualified access to the supported interface templates of an object template or an object group template in a required interface specification.

Having introduced some of the basic constructs of the ITU-ODL language, it will now examined how ITU-ODL specifications can be presented as Recommendations, particularly with a view to reusing ITU-ODL (or ODP-IDL) specifications.

5.3 Interface template, object template and object group template separation and sharing

Freedom is offered to the developer of computational specifications for independent declaration of interface templates, object templates and object group templates. Each interface template in ITU-ODL may be reused in any number of object templates. Similarly, object templates may be specified as individual definitions, and reused in any number of object group templates.

As ITU-ODL is a superset of ODP-IDL; the mapping between an ITU-ODL operational interface template and an equivalent ODP-IDL specification is trivial. An advantage of such a straightforward mapping lies in the ability to use existing CORBA-based tools in the software development chain. An additional advantage is the ability to reuse existing ODP-IDL interface definitions.

As seen above, an advantage of separating construct declarations is that it provides a straightforward means of sharing construct declarations. Below, the principles of sharing are presented in more detail.

5.3.1 Data types

(R10) Data types can be declared in any ITU-ODL scope. Sharing of data type declarations between several operations or flows of differing interface templates is allowed.

5.3.2 Operations

(R11) Operation signatures are declared within interface templates. Because operation signatures are not declared separate from interface templates, and there is no specific suitable sharing mechanism, sharing of an operation signature declaration between several interface templates is not possible.

NOTE – This rule is directly derived from ODP-IDL. A new version of ODP-IDL may relax this rule. The implication of such a change for ITU-ODL is an open issue.

(R12) Two operations with the same identifier declared in two distinct interface templates are considered different.

5.3.3 Flows

(R13) Flow signatures are declared within interface templates. Because flow signatures are not declared separate from interface templates, and there is no specific suitable sharing mechanism, sharing of a flow signature declaration between several interface templates is not possible.

(R14) Two flows with the same identifier declared in two distinct interface templates are considered different.

5.3.4 Interface templates

It is assumed that interface template declaration sharing is intended for the sharing of an ITU-ODL specification of an interface template between several object templates. ITU-ODL syntax is defined to enable the separation of interface template declarations from object template declarations. Interface template specifications can be included in an object template declaration as supported interfaces or as required interfaces.

5.3.4.1 (D4) declared supported interfaces/offered interfaces: Interface templates listed as being supported on an object template are the only interface templates for which instances may exist on the object's instance[2]. The offered interfaces of an object instance are the instances of interfaces existing on that object at a particular time.

5.3.4.2 (D5) declared required interfaces: The declared required interfaces on an object template list the interface templates which an instance of the object template needs to invoke operations upon[3].

The following rules deal with the relation of operation, flow, interface template and object template declarations:

(R15) Operation and flow signatures are only declared within interface templates. Interface templates can be declared inside and outside object templates and, inside and outside object group templates.

5.3.5 Object templates

It is assumed that object template declaration sharing is intended for the sharing of an ITU-ODL specification of an object template between several group templates. ITU-ODL syntax is defined to enable the separation of object template declarations from group template declarations. Object template specifications can be referred in a group template as members. Interface template specifications can be referred in an object group template as supported or required contracts, which are the interface templates which instances can be used by entities external to the object group (supported) or needed by the instances of the group members from the environment (required).

5.3.5.1 (D6) declared members: The declared members of an object group template are the object templates, or object group templates, belonging to that group template. The declared members are listed as "members" on a group template.

5.3.5.2 (D7) declared required/supported contract: A declared required/supported contract of an object group template is one of the required/supported interfaces of a member object template or group template of that object group template. The declared required/supported contracts on an object group template represent the only interfaces able to be used (supported) by entities external to that

---

[2] The declared supported interfaces of a base class are considered as supported interfaces of the subclass and may be instantiated by the object instance of the subclass template as well.

[3] The declared required interfaces of a base class are considered as required interfaces of the subclass.

object group or which the instances of the members can be accessed in the environment (required). The declared contracts are listed as required or supported on a group template (similar to object templates). If no required and supported interfaces are specified in a group template, no restrictions concerning the visibility of interfaces of the group members are applied[4].

The rule is as follows:

(R16) Object templates can be declared outside object group templates.

5.3.6 Scoping rules

The following scoping rules are relevant to shared specifications.

(R17) In the scope of an object template, an interface identifier can be defined by declaring the associated interface template in-line, or used by declaring it as supported or required. In each case, the global name of the interface template will be different: something like ...<object-identifier>::<interface-identifier> in the former case and something like ...<interface-identifier> in the latter case.

(R18) In the scope of an object group template, an object identifier can be defined by declaring the associated object template in-line, or used by declaring it as a member. In each case, the global name of the object template will be different.

(R19) In the scope of an object group template, an interface identifier can be defined by declaring the associated interface template in-line, or used by declaring it as a required/supported contract. In each case, the global name of the interface template will be different.

5.4 Behaviour

The behaviour of an entity (interface, object, object group template), in its most general sense, consists of all the possible interactions the entity can undertake with its environment. ITU-ODL is not sufficiently mature to provide a complete and detailed specification for behaviour in this sense. Instead, an informal behaviour specification is described for particular entities as follows:

Interface templates

This specification describes the service provided by an instance of the template being defined. It also describes the intended usage of the interface. This specification documents the ordering (or sequencing) constraints on the operations defined in the interface template. Invocations of operations on an instance of such a template must satisfy these constraints. In the current version of ITU-ODL, this specification is a string literal.

Object templates

This specification describes the responsibilities of an object in providing services via each of its interfaces as supported in ITU-ODL.

Object group templates

This specification describes the criterion which holds for all entities contained in the group template. Depending on the criteria the group predicate specification may contain additional information. For example if the criteria is that the members of the group perform together a particular service, the predicate description can additionally define functionality provided on each supported contract.

---

[4] This is due to the fact that different criteria for defining groups are possible and for some of them the specification of contracts is not useful.

NOTE – It should be noted that the string provided as a behaviour specification can be a reference to a formal behaviour specification done in another language. An example could be a link from an operational interface template definition to a SDL [7] process type specification which contains a formal behaviour specification for that interface template. Specification and interpretation of those links is subject to CASE tools using ITU-ODL as a description technique for computational viewpoint specifications.

5.5 Inheritance

5.5.1 Introduction and motivation

Interface template, object templates and object group templates provide specification modularity. Since these three templates also represent types, it is convenient to define a reusability mechanism where:

- a construct can directly rely on the entities defined in another construct (of the same kind of template). For example, a data type defined in one interface template is used within another interface template;
- a construct is derived from another construct (of the same kind of template). For example, one object template specification is derived from another object template specification.

In classic object-based literature, such a reuse mechanism is known as inheritance. In the case of ITU-ODL, defining rules for inheritance will allow new interface templates, object templates and object group templates to be declared as extensions or restrictions of previously defined ones.

The remainder of this clause describes the principles underlying ITU-ODL's reuse of specifications through inheritance. First, the essential elements of interface template inheritance, object template inheritance and object group template inheritance are presented. This is then followed by an explanation of naming and scoping rules related to inheritance.

5.5.2 Definitions

The following definitions are relevant to inheritance.

5.5.2.1 (D8) base/derived/specialized construct: A construct (group template, object template or interface template) is said to be derived from or specializing another construct, called a base construct of the derived construct, if it inherits from this base construct.

5.5.2.2 (D9) most specialized: The most specialized object templates (group templates or interface templates) within a set of object templates (group templates or interface templates) are the elements of the set from which no other constructs within the set are derived (i.e. which are not the base of any other construct of the set).

5.5.2.3 (D10) direct/indirect base: A construct is called a direct base of a construct if it is mentioned in the inheritance specification of the construct declaration, and an indirect base if it is not a direct base but the base of a direct or indirect base (sub-inheritance).

5.5.2.4 (D11) inheritance graph/partial inheritance graph: The inheritance graph of object templates (group templates, or interface templates) is the directed acyclic graph representing the inheritance relationships between object templates (group templates or interface templates). A partial inheritance graph of a given type is an inheritance graph restricted to a set of constructs.

NOTE – The leaves of the inheritance graph for a given type of construct (i.e. for group templates, object templates or interface templates) are the most specialized constructs of this type.

5.5.2.5 (D12) restriction/restricted set: The restriction of a set of constructs is constructed by removing from this set any construct which is the base of any other construct from the set. The result of the restriction of a set is called a restricted set.

NOTE 1 – A restricted set can also be seen as the set of leaves of the partial inheritance graph.

NOTE 2 – The restricted set of all the constructs of type object template (group template or interface template) is the set of most specialized object templates (group templates or interface templates).

Assume that the following interface templates with the following inheritance relationship are defined:

- interface template I1;
- interface template I2, which inherits from interface template I1;
- interface template I3, which inherits from interface template I1;
- interface template I4.

The restriction of the set of interface templates (I1, I2, I3, I4) is the set (I2, I3, I4).

5.5.3 Inheritance in construct declarations

5.5.3.1 Interface template inheritance

It is assumed that interface template inheritance provides "reuse by specialization" of (an ITU-ODL specification of) an interface template. The interface template that is inherited from is called a base interface template. The interface template that does the inheriting is called the derived interface template. This specialization can take two forms:

- addition of new operations, or flows, to the list of the base interface templates;
- redefinition of the signatures of operations or flows in the base interface templates.

NOTE 1 – The current version of ODP-IDL does not allow this kind of inheritance for operational interface templates (as reflected in rule 24); hence, neither does ITU-ODL.

NOTE 2 – All derived interface templates of a base interface template are considered "compatible with" the base interface template.

The syntax defined for ITU-ODL fully supports ODP-IDL interface (template) inheritance rules, and adopts consistent rules for both operational and stream interface templates. The ITU-ODL interface template inheritance rules are as follows:

General rules

(R20) An interface template can be derived from one or several other interface templates, each of which is called a base interface template of the derived interface template. In the case of derivation from multiple base interfaces (multiple inheritance), the order of derivation is not significant.

(R21) An interface template may not be specified as a direct base interface template of a derived interface template more than once. It may be an indirect base interface template more than once (i.e. a "diamond shape" inheritance graph is possible).

(R22) A derived interface template may declare new sub-constructs (data types, operations or flows). Unless redefined, the sub-constructs of the base interface template can be referred to as if they were sub-constructs of the derived interface template. Interface template inheritance causes all identifiers in the closure of the inheritance graph to be imported into the current naming scope.

(R23) It is illegal to inherit from two interface templates having the same operation identifiers, or having the same flow identifiers.

(R24) It is illegal to redefine an operation in the derived interface template.

(R25) It is illegal to redefine a flow in the derived interface template.

(R26) A derived interface template may redefine data type identifiers inherited. A data type identifier from an enclosing scope can be redefined in the current scope.

Behaviour

(R27) The behaviourText of base interface templates is not available in a derived interface template.

(R28) The usage attribute of base interface templates is not available in a derived interface template.

As an illustrative example, assume that object template O4 declares as supported:

– interface template I4, a specialization of interface template I1 constructed by the addition of operation41; and

– interface template S2, a specialization of interface template S1, with the addition of source videoFlow21.

It is possible to declare interface template I4 inheriting operation11 from I1, and adding operation operation41. Similarly, S2 may inherit from S1 the source flow voiceDownStream and the sink flow voiceUpStream, and add the source flow videoFlow21.

I1 and S1 are defined as follows:

```
interface I1{
        ...
        // data types
        typedef … DataType11;
        typedef … DataType12;

        void operation11 (in DataType11 …, out DataType12 …);
}; // end of I1

interface S1{
        ...
        // flow types
        typedef … VoiceFlowType;

        source VoiceFlowType voiceDownStream;
        sink VoiceFlowType voiceUpStream;
}; // end of S1
```

The inheriting interface templates can then be defined as follows:

```
interface I4: I1{
        ...
        typedef … DataType41;
        void operation41 (in DataType41 …);
}; // end of I4

interface S2: S1{
        ...
        typedef … FlowTypeS21;
        source FlowType21 videoFlow21;
}; // end of S2
```

Object template O4, using the inherited specialized interface templates, can be defined as follows:

```
CO O4{
        behaviour
        ...
        supports
                I4, S2;
        ...
}; // end of O4
```

5.5.3.2 Object template inheritance

It is assumed that object template inheritance is intended to provide "reuse by specialization" of (an ITU-ODL specification of) an object template. The object template that is inherited from is called a base object template. The object template that is doing the inheriting is called the derived object template. This specialization can take either of two basic forms:

- addition of interface templates: new interface templates may be added to the list of supported/required interfaces of the base object;
- refinement of interface templates: supported interfaces on the base objects may be specialized in the derived object.

The inheritance rules for object templates are as follows:

General rules and supported interfaces

(R29) An object template can be derived from one or several other object templates, each called a base object template of the derived object template. In the case of derivation from multiple base object templates (multiple inheritance), the order of derivation is not significant.

NOTE 1 – The inheritance graph for object templates is completely separated from the inheritance graph for interface templates.

(R30) A derived object template may declare new sub-constructs (data types, interface templates). Unless redefined, the sub-constructs of the base object template can be referred to as if they were sub-constructs of the derived object template. Object inheritance causes all identifiers in the closure of the inheritance graph to be imported into the current naming scope.

(R31) An object template may not be specified as a direct base object template of a derived object template more than once. It may be an indirect base object template more than once ("diamond shape" inheritance graph).

(R32) The interface templates which may be offered on a derived object is the union of the interface templates supported on all of the base objects, plus any additional interface templates declared supported on the derived object template.

NOTE 2 – To add a new interface template to the list of interface templates inherited from the base object templates, it is sufficient to declare an additional supported interface template in the object template (addition of interface).

NOTE 3– To refine an interface template supported by an object's base object templates, it is sufficient to declare a supported interface template in the object template, where that supported interface template is derived from the former one (refinement of interface template). The object may then offer instances of either of these interface templates.

(R33) A derived object template may redefine data type identifiers inherited. A data type identifier from an enclosing scope can be redefined in the current scope.

Behaviour

(R34) The behaviourText of base object templates is not available in a derived object templates.

(R35) The required interfaces of the base object template is the union of the required interfaces of the base object templates, plus any additional required interfaces specific to the derived object template.

Initial

(R36) The initial interfaces of base object templates are not available in a derived object template; initial interfaces are not inherited.

The initial interface template of a derived object template must be derived from (or may be identical to, in the case of single object template inheritance) the initial interface templates of direct base object templates. Otherwise, a management system (for instantiation) will have difficulty seeing the derived object templates as equivalent to its base templates.

As an illustrative example, consider object template O6 which supports the following:

- interface template I1 which is supported by object templates O1 and O2;
- interface template I5 supporting operation Op21 which is also defined on interface template I2 supported by O1, and in addition Op51;
- interface template I3 supported by O2;
- interface I6, supporting operation Op61.

The following inheritance relationships can be established between object templates O1, O2, and O6 (see Figure 5):

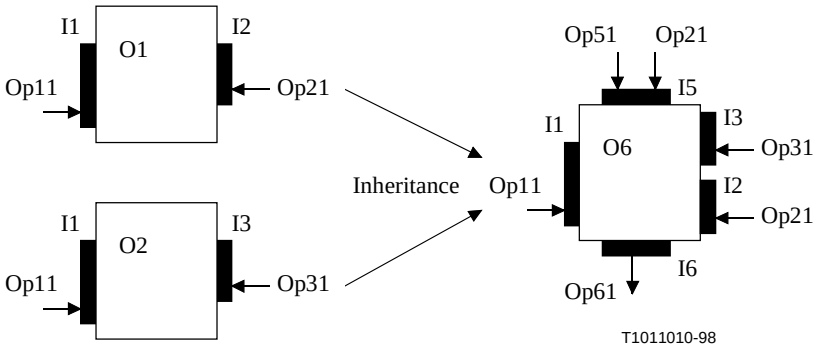

Figure 5/Z.130 – Inheritance between object templates

In this case, the definition of O6 can be constructed from O1 and O2 with the aid of inheritance rules, and the addition of new members.

Interface template I5 is declared as derived from interface I2, and interface template I6 is declared in isolation:

```
interface I5: I2{
        ...
        typedef … DataType51;
        void Op51(in DataType51 …);
}; // end of I5
```

```
interface I6{
        ...
        typedef … DataType61;
        void Op61(in DataType61 …);
}; // end of I6
```

Object template O6 is declared as derived from object templates O1 and O2, and interface templates I5 and I6 are declared in the list of supported interface templates of O6:

```
CO O6: O1, O2{
        ...
        supports
                I5, I6;
        ...
}; // end of O6
```

Since there is no relationship of inheritance from interface template I6 with any interface template declared in the base object templates O1 or O2, interface template I6 is simply considered a supported interface of O6.

Interface template I3 and I2 appears on O2, and through the rules of inheritance is also considered as a supported interface of O6.

Interface template I1 appears on both O1 and O2, and through the rules of inheritance is also considered as a supported interface of O6.

Interface template I5 inherits from I2 and offers both operation Op21 and Op51. Note, that the object can instantiate the basetype (I2) as well as the subtype (I5).

5.5.3.3 Object group template inheritance

It is assumed that group template inheritance provides "reuse by specialization" of (an ITU-ODL specification of) a group template. The group template that is inherited from is called a base group template. The group template that does the inheriting is called the derived group template. This specialization can take either of two basic forms:

- addition of object templates/group templates: new object templates may be added to the list of members of the base group template;
- refinement of object templates/group templates: members on the base group templates may be specialized in the derived group templates.

The inheritance rules for group templates are as follows:

General rules and members

(R37) A group template can be derived from one or several other group templates, each called a base template of the derived group template. In the case of derivation from multiple base group templates (multiple inheritance), the order of derivation is not significant.

NOTE 1 – The inheritance graph for object templates is completely separated from the inheritance graph for object and interface templates.

(R38) A derived group template may declare new sub-constructs (data types, interface templates, object templates, group templates). Unless redefined, the sub-constructs of the base group template can be referred to as if they were sub-constructs of the derived group template. Group template inheritance causes all identifiers in the closure of the inheritance graph to be imported into the current naming scope.

(R39) A group template may not be specified as a direct base group template of a derived group template more than once. It may be an indirect base group template more than once ("diamond shape" inheritance graph).

(R40) The object/group templates which may comprise a derived group template is the union of the member object/group templates declared on all of the base group templates, plus any additional object/group templates declared supported on the derived group template.

NOTE 2 – To add a new object/group to the list of member templates inherited from the base group templates, it is sufficient to declare an additional member object/group template in the group template (addition of object/group).

NOTE 3 – To refine an object/group template supported by a group's base group templates, it is sufficient to declare a member object/group template in the group template, where that member object/group template is derived from the former one (refinement of object/group templates). The group may then include instances of either of these object/group templates.

(R41) A derived group template may redefine data type identifiers inherited. A data type identifier from an enclosing scope can be redefined in the current scope.

Behaviour

(R42) The predicates of base groups are not available in a derived group. The predicate of the derived group is defining the criteria which all members of the group fulfil.

Contracts

(R43) The required/supported contracts which comprise a derived group template is the union of the required/supported contracts comprising the base group templates, plus any additional required/supported contracts declared on the derived group template.

If the predicates of the subgroup are different from the base template, contracts can still be inherited, if their definitions are compatible.[5] The user should avoid to use inheritance if the predicates of base group templates and subgroup templates are not compatible.

5.5.3.4 Naming and scoping with respect to inheritance

The following scoping rules are added to support inheritance capabilities:

(R44) Inheritance introduces identifiers into the derived interface template, object template or object group template.

(R45) Inheritance of interface templates, object templates or object group templates introduces multiple global ITU-ODL identifiers for the inherited identifiers.

(R46) A qualified name (one of the form <scoped-name>::<identifier>) is resolved by locating the definition of <identifier> within the scope. The identifier must be defined directly in the scope or (if the scope is an object group template, object template or interface template) inherited into the scope. The identifier is not searched for in enclosing scopes.

---

[5] The definition of predicate compatibility is not subject of this Recommendation.

6 Specification of ITU-ODL

This clause defines the syntax of ITU-ODL. It is divided into four parts which deal with the major constructs of ITU-ODL:

- type and constant declaration;
- interface templates;
- object templates;
- object group templates.

6.1 Type and constant declaration

6.1.1 Structure

Data types and constants can be declared in almost any scope within an ITU-ODL specification. These types or constants can be used for declaration of operation, exception, flow, and other template constructs. As for any template declaration, it is required that a type or constant be declared prior (i.e. earlier in the file) to its use.

The syntax supported by ITU-ODL for type and constant declaration is identical to the one of ODP-IDL. The reader can find in Appendix II a description of this syntax.

6.1.2 Example of type and constant declarations

The following example shows the declaration of three data types: Bps, which is a synonym for float; Guarantee, which is an enumeration; and AudioQoS, which is a structure.

```
typedef float Bps;

enum Guarantee {
        Deterministic,
        Statistical,
        BestEffort
};
struct AudioQoS {
        union Throughput switch (Guarantee){
                case Statistical:       Bps   mean;
                case Deterministic:     Bps   peak;
                case BestEffort:        range struct Interval {
                                                Bps min;
                                                Bps maxd;
                                        };
        };
        union Jitter switch (Guarantee) {
                case Statistical:       Bps   mean;
                case Deterministic:     Bps   peak;
        };
};
```

6.2 Interface template

6.2.1 Structure

A computational interface template comprises:

- a (textual) behaviour specification;

  and, as appropriate, either:

- an operational interface signature; or
- a stream interface signature.

This structure is reflected in the ITU-ODL rule for <interface_body>. The subsequent subclauses of this Recommendation examine the elements of this structure in more detail.

The following syntax is defined for interface template declaration:

```
<interface_template>        ::=  <interface_header> "{" <interface_body> "}"
<interface_header>          ::=  "interface" <identifier>
                                 [ <interf_inheritance_spec> ]
<interf_inheritance_spec>   ::=":" <scoped_name> { "," <scoped_name> }*
<interface_body>            ::=  [<interf_behaviour_spec>]
                                 { <op_sig_defns> | <stream_sig_defns> }
<interf_behaviour_spec>     ::=  "behaviour" {
                                 {<interf_behaviour_text> [<interf_usage_spec>]}
                                 | <interf_usage_spec> }
<interf_behaviour_text>     ::=  "behaviourText" <string_literal> ";"
<interf_usage_spec>         ::=  "usage" <string_literal> ";"
```

6.2.2 Interface template inheritance

The rules which apply to interface template inheritance are those specified for ODP-IDL, with extensions to deal with flows.

The following syntax is defined for interface template inheritance:

```
<interface_header>          ::=  "interface" <identifier> [ <interf_inheritance_spec> ]
<interf_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
```

It should be noted that the ODP-IDL specification in its current form prohibits redefinition of an operation identifier in a derived interface template specification and prohibits inheritance of two operations of the same identifier. Initially, ITU-ODL will be restricted according to this limitation of ODP-IDL in operational and stream interface template definitions.

Consistent with ODP-IDL, interface template inheritance is the equivalent of simple inclusion of all attributes (including interface attributes), operations and flows from the base interface template into the derived interface template. This inclusion involves all attributes, operations and flows of the base interface template, including those obtained by inheritance from other interface template specifications. Hence a derived template should always be capable of providing the services of the base interface template.

A stream interface template cannot inherit from an operational interface template and vice versa.

6.2.3 Interface template behaviour specification

The interface signature describes only the syntactic structure of an interface template. Signature compatibility is less discerning than behaviour compatibility. It is indeed possible that two interfaces have compatible signatures but differ completely in their behaviour. This clause describes how at least a textual behaviour is specified in ITU-ODL.

The following syntax is defined for the interface template behaviour specification:

```
<interf_behaviour_spec> ::=   "behaviour" {
                                    {<interf_behaviour_text> [<interf_usage_spec>]}
                                    | <interf_usage_spec> }
<interf_behaviour_text>  ::=  "behaviourText" <string_literal> ";"
<interf_usage_spec>      ::=  "usage" <string_literal> ";"
```

6.2.4 Operational interface signature

An operational interface signature comprises a set of interrogation and announcement signatures, one for each operation type in the interface template. An operational interface signature specifies the following information (similar to ODP-IDL):

- an optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked (interrogation or announcement);
- the type of the operation return result (void otherwise);
- the operation identifier;
- a parameter list (zero or more parameters to the operation);
- an optional "raises" expression which indicates which exceptions may be raised as a result of an invocation of this operation.

The following syntax is defined for the signature of operational interface templates. It is similar to the ODP-IDL syntax for interface (template) declaration:

```
<op_sig_defns>          ::=  { <op_sig_defn> ";" }*
<op_sig_defn>           ::=  { <announcement> | <interrogation>
<announcement>          ::=  "one-way" "void" <identifier> <parameter_dcls>
<interrogation>         ::=  <attr_dcl>
                        |    <oper_dcl>
<attr_dcl>              ::=  ["readonly"] "attribute" <param_type_spec>
                             <declarators>
<oper_dcl>              ::=  <op_type_spec> <identifier>
                             <parameter_dcls>
                             [<raises_expr>] [<context_expr>]
<op_type_spec>          ::=  <param_type_spec>
                        |    "void"
<parameter_dcls>        ::=  "(" <param_dcl> { "," <param_dcl> }* ")"
                        |    "(" ")"
<param_dcl>             ::=  <param_attribute> <param_type_spec>
                             <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>
                        |    <scoped_name>
```

6.2.5 Operational interface attributes

Operational interface attributes are logically equivalent to defining a pair of accessor functions: one to set the value of the attribute and one to get the value of the attribute.

6.2.6 Stream (flow) signature

A stream interface template is comprised of a set of flow types. Each flow type contains the identifier of the flow, the information type of the flow, and an indication of whether it is a producer or consumer (but not both) with respect to the object which provides the service defined by the template.

The syntax defined here presupposes a directionality with respect to the stream interface template definitions. If two objects are involved in a stream binding, then one is designated a service provider, or server, and the other a service consumer, or client. The interface template describing interactions between them is expressed from the viewpoint of the client (defining the server). In many ways, particularly where flows travel in both directions, the choice of client and server may appear rather arbitrary. However, this model is consistent with many familiar service models. Note that the server template includes a declaration that it "supports" the stream interface template, while the client template includes a declaration that it "requires" the stream interface template.

For example, in order to play a video game, a client (the Player) locates an appropriate interface (VideoGame) to the server (the Game) (see Figure 6). The service is defined naturally in terms of information sources (the video and audio) and sinks (the controls labelled joystick1 and joystick2). However, all of these definitions presuppose a directionality, or point of view, namely that of the client. The service view held by the game itself, involving a source of control functions and a sink of video and audio information, can easily be obtained from the other definition via a simple mapping. As a result, one of these service definitions is redundant. Stubs for either the Player object (as a client) or the Game object (as a server) may be produced from a single interface template specification.

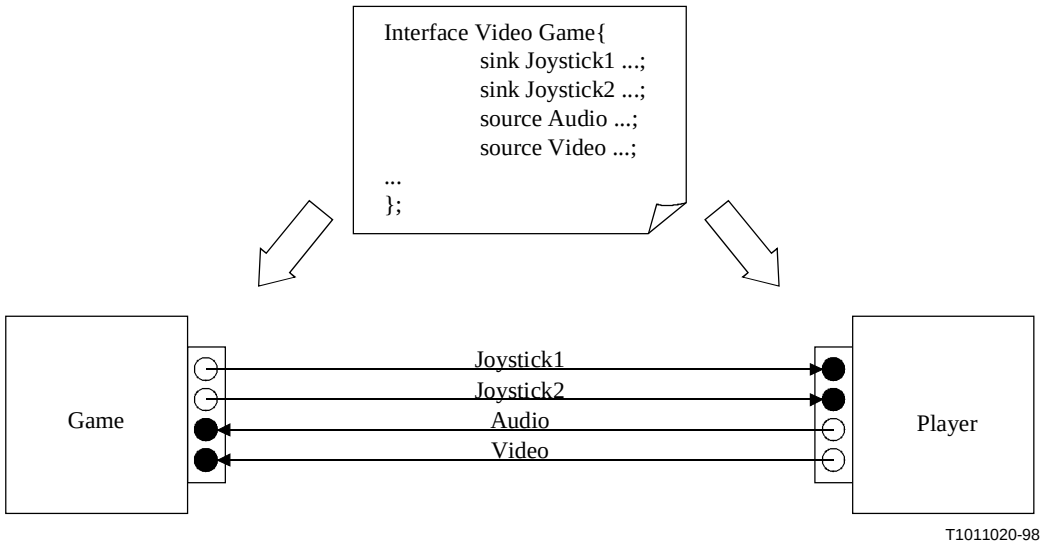

Figure 6/Z.130 – Stream interface template example

In ITU-ODL, stream interface templates are defined as the client's view on the server. Each flow is specified as a source if information flows from the server to the client, and as a sink if it flows in the opposite direction. In the object template definition of the server, the stream interface template is listed as a supported interface, while on the client, the interface template is listed as a required interface.

```
<stream_sig_defns>          ::=   { <stream_flow_defn> ";" }*
<stream_flow_defn>          ::=   <flow_direction> <flow_type>
                                  <identifier>
<flow_direction>            ::=   "source" | "sink"
<flow_type>           ::=   <param_type_spec>
```

6.2.7 Example of interface template declaration

Below is an example of an interface template CSMConfiguration that is derived by inheritance from an interface template ServiceManagement as defined in the module Management. It contains operation definitions, attribute definitions and a behaviour definition.

```
interface CSMConfiguration: Management:: ServiceManagement {
    behaviour
        behaviourText
        "This interface serves to configure the co CSM.

        The ReadState operation returns a complete
        representation of the CSM state. The WriteState operation
        allows the complete CSM state to be set.";

        usage
        "Operation init must be invoked prior to other
        operations defined on the service."
```

6.3 Object template

6.3.1 Structure

An object template specification comprises two high-level parts. The first supports inheritance, and is associated with the declaration of the object template's identifier. The second part is the object template body, which comprises the main sub-parts of the template as follows:

- a behaviour specification;
- a specification of required interfaces;
- a specification of supported interfaces; and
- an initialization specification.

Below, these specifications are examined in more detail, as well as the syntax that supports inheritance.

The following syntax is defined for object template declaration:

```
<object_template>           ::=   <object_template_header>
                                  "{" <object_template_body> "}"
<object_template_header>    ::=   "CO" <identifier>
                                  [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=   ":" <scoped_name> { "," <scoped_name> }*
<object_template_body>      ::=   [<supporting_def_spec>]
                                  [<interface_def_spec>]
                                  [<object_behaviour_spec>]
                                  [<reqrd_interf_templates>]
                                  <suptd_interf_templates>
                                  [<object_init_spec>]
```

6.3.2 Object template inheritance

Object template inheritance is intended to support specification reuse and to provide a mechanism for defining compatibility via sub-typing relationships.

The following syntax is defined for object template inheritance:

```
<object_template_header>    ::=  "CO" <identifier>[ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
```

Object template inheritance is the equivalent of simple inclusion of all constraint attributes, type definitions, required and supported operational and stream interface template specifications from the base object templates into the derived object template. This inclusion involves all attributes, types and interface templates of the base object template, including those obtained by inheritance from other object template specifications.

There are no restrictions on the names of interface templates or types inherited from ITU-ODL base object templates.

The initial interface specified in a derived object template must be of a type which is the same as, or derived from, all initial interface templates of the corresponding base object templates.

6.3.3 Object template behaviour specification

The behaviour of an object is specified as a string in the object template, that should describe the role of an object in providing services via each of its interfaces.

The following syntax is defined for object template behaviour specification:

```
<object_behaviour_spec>     ::=  "behaviour"
                                 <string_literal> ";"
```

6.3.4 Required interface templates

The second comprises the (declared) required interfaces which specifies interface templates used by instances of the object template to perform their functions and provide their services.

It is possible to address a required interface via <tagged_name> directly from a particular object template. In that case, the specified object template must have a supported interface template, which is compatible to the specified required interface template. The compatibility rules are open; an example are the rules provided in [2] and [3]. This notation ensures that compatibility between the required and supported interfaces can be checked on this static specification level.

The following syntax is defined for required interface template definition:

```
<reqrd_interf_templates>    ::=  "requires" <req_interf_defn>
                                 {"," <req_interf_defn> }* ";"
<req_interf_defn>           ::=  <scoped_name> | <tagged_name>
```

6.3.5 Supported interface templates

The (declared) supported interfaces of an object template are the interfaces listed as supported in the template specifications. Instances of interface templates declared as supported may be offered by instances of templates being defined.

The following syntax is defined for supported interface declaration:

```
<suptd_interf_templates>    ::= "supports" <suptd_interf> ";"
<suptd_interf>              ::= <suptd_interf_defn> {"," <suptd_interf_defn> }*
<suptd_interf_defn>         ::= <scoped_name> |     <interface_template>
```

6.3.6 Object template initialization specification

The initialization specification identifies an interface template, a reference to which will be returned to the instantiator of the object template being defined. This interface may be used to initialize the newly instantiated object. It should be noted that the initial interface is also one of the supported operational interfaces[6].

The following syntax is defined for the initialization specification:

<object_init_spec> ::= "initial"
{ <scoped_name> | <interface_template> } ";"

6.3.7 Example of object template declaration

Following is an example showing how an object template is declared. It begins with the keyword "CO", which is then followed by the identifier of the object template, CSMfactory. This template does not inherit from any other, as indicated by the absence of any inheritance specifications. The body of the template is then declared between the braces.

```
CO CSMfactory {
        requires
                QoSmanagerIF;

        supports
                Management,
                LcgFactory,
                CSMConfiguration;

        initial
                Management;

}; // end CSMfactory
```

The interface template MyManagement inherits from the initial interface of the base object template. It should be noted that the derived object template supports an interface template, MyCSMConfiguration, which is derived from an interface template supported by the base template, CSMConfiguration. The instantiation of either or both of these types at any particular time is an implementation decision. An implementation of the MyCSMfactory object template may instantiate zero or more instances of CSMConfiguration for example, and still conforms to this specification. The number of instances of any particular interface template may be constrained by the object template behaviour specification.

```
interface MyManagement: Management{
        ...
}; // end MyManagement


interface MyCSMConfiguration: CSMConfiguration{
        ...
}; // end MyCSMConfiguration
```

[6] The initial interface does not need to be included in the supported interface definition clause.

```
CO MyCSMfactory: CSMfactory{

        requires
                AccountingEventIF;
        supports
                MyManagement,
                MyCSMConfiguration;

        initial
                MyManagement;

}; // end MyCSMfactory
```

An example for a behaviour specification is given below.

```
CO Timer{

        behaviour
                "Instances of this co periodically call the
                tick function of a specified TimerInterrupt
                interface.";

        requires
                TimeInterrupt;
...
};
```

6.4 Object group template

6.4.1 Structure

A group template specification comprises two high-level parts. The first supports inheritance, and is associated with the declaration of the group template's identifier. The second part is the group template body, which comprises the main sub-parts of the template as follows:

- a behaviour specification;
- a specification of contained object templates and object group templates; and
- a specification of interfaces templates, which instances are visible outside the group.

Below, these specifications are examined in more detail, as well as the syntax that supports inheritance.

The following syntax is defined for object group template declaration:

```
<group_template>          ::=  <group_template_header>
                               "{" <group_template_body> "}"
<group_template_header>   ::=  "group" <identifier>
                               [ <group_inheritance_spec> ]
<group_template_body>     ::=  [<supporting_def_spec>]
                               [<interface_def_spec>]
                               [<object_def_spec>]
                               [<group_def_spec>]
                               [<group_predicate_spec>]
                               <supp_comp_templates>
                               [<supported_contract_interfaces>]
                               [<required_contract_interfaces>]
```

6.4.2 Object group template inheritance

Object group template inheritance is intended to support specification reuse and to provide a mechanism for defining compatibility via sub-typing relationships. The purpose of such compatibility for object template specifications is to support the use of object framework specifications.

The following syntax is defined for group inheritance:

```
<group_template_header>     ::=   "group" <identifier> [ <group_inheritance_spec> ]
<group_inheritance_spec>    ::=   ":" <scoped_name> { "," <scoped_name> }*
```

Group template inheritance is the equivalent of simple inclusion of all type definitions, contracts and member specifications from the base group templates into the derived group template. This inclusion involves all attributes, types and object templates of the base group, including those obtained by inheritance from other group template specifications.

There are no restrictions on the identifiers of object templates or types inherited into group templates.

6.4.3 Object group template predicate specification

The predicate specifications of an object group template has the purpose to identify the criteria, which all members of the group fulfil. The range of possible criteria is open. Examples include:

- structuring purpose;
- management issues (domain membership, appliance of same policies);
- implementation issues (group members together perform a particular service).

Depending on the criteria, the group template predicate specification may contain any additional information which is useful in the context.

The following syntax is defined for group predicate specification:

```
<group_predicate_spec>      ::=   "predicate" <string_literal> ";"
```

6.4.4 Member object templates and group templates

Member object templates and object group templates are the object templates and object group templates that belong to the object group template.

NOTE – An object or group can be contained in more than one group.

Following is the syntax supporting member object templates:

```
<supp_comp_templates>       ::=   "members" <suptd_comp> ";"
<suptd_comp>                ::=   <suptd_comp_defn> {"," <suptd_comp_defn> }*
<suptd_comp_defn>           ::=   <scoped_name>
                            |     <object_template>
                            |     <group_template>
```

6.4.5 Contracts

Contracts are the interfaces of the members of the object group that are visible to entities outside the object group. Members can only be accessed from the environment via these interfaces. In the object group template specification they are indicated as a simple list of (optionally scoped) identifiers.

Following is the syntax supporting contracts:

```
<supported_contract_interfaces>     ::= "supports" <scoped_name> { "," <scoped_name> }*  ";"
<required_contract_interfaces>      ::= "requires" {<scoped_name> | <tagged_name>}
                                          { "," {<scoped_name> | <tagged_name>} }*  ";"
```

6.4.6 Example of group template declaration

Below is an example showing how a group template is declared. The example presented is an object group template subnetManager.

```
interface Configuration {...};
interface Configurator {...};
interface Trail {...};
interface TC {...};

CO CMC {

        requires
                Configuration;
        supports
                Configurator;
        ...
};
CO NetworkCoordinator {
        requires
                TC, SncService, SncServiceFactory;
        supports
                Trail, TC, Configuration;
        ...
};
CO NetworkCP {
        supports
                SncService, SncServiceFactory, Configuration;
        ...
};
CO ElementCP {...};


group SubnetManager {
        predicate
                "This group manages a subnetwork"

        members
                CMC, NetworkCoordinator, CMC, NetworkCP, ElementCP;

        supported
                Configurator, Trail, TC;
};
```

ANNEX A

BNF

A.1 Compliance

A reference acceptor for ITU-ODL is available at [9]. Where ambiguities exist between the text in this Recommendation and the reference acceptor, the text takes precedence.

NOTE – An acceptor is a tool that computes whether a given specification is ITU-ODL-compliant or not.

A.2 Lexical conventions

ITU-ODL uses the lexical and pre-processor conventions of ODP-IDL [8].

A.3 Keywords

Most keywords are imported from ODP-IDL but several are added to support the extensions of ITU-ODL with respect to ODP-IDL. These keywords are underlined.

any	attribute	behaviour	behaviourText	boolean	case
char	members	const	context	default	double
enum	exception	FALSE	fixed	float	group
in	initial	inout	interface	long	module
Object	CO	octet	one-way	out	predicate
raises	requires	readonly	sequence	short	sink
source	string	struct	supports	switch	TRUE
typedef	unsigned	union	usage	void	wchar
wstring					

A.4 Extended BNF notation

The following meta-symbols are used to describe ITU-ODL's syntax. The description is an extended Backus-Naur Form (BNF)[7].

Symbol	Meaning
`::=`	Defined to be
`|`	Alternatively
`<text>`	non-terminal
`"text"`	terminal (i.e. literal)
`*`	the preceding syntactic unit may be repeated zero or more times
`+`	the preceding syntactic unit may be repeated one or more times
`{}`	the enclosed syntactic units are grouped as a single syntactic unit
`[]`	the enclosed syntactic unit is optional – may occur zero or one time

A.5 Syntax

The syntax for ITU-ODL is presented below. Expressions taken from ODP-IDL are marked with a*.

Top level syntax

```
<odl_spec>          ::=  <definition>*
<definition>        ::=  <module> ";"
                    |    <group_dcl> ";"
                    |    <object_dcl> ";"
                    |    <interface_dcl> ";"
                    |    <supporting_def> ";"
```

[7] Note that concatenation of symbols has a higher precedence than |. For example, X X X | Y Y Y is equivalent to {X X X} | {Y Y Y}.

A.5.1 Module syntax

```
<module>                    ::=  "module" <identifier> "{" <definition>+ "}"
<scoped_name>               ::=  <identifier>
                            |    "::"<identifier>
                            |    <scoped_name> "::" <identifier>
<tagged_name>               ::=  <scoped_name> "." <scoped_name>
```

A.5.2 Group syntax

```
<group_dcl>                 ::=  <group_forward_dcl>
                            |    <group_template>
<group_forward_dcl>         ::=  "group" <identifier>
<group_template>            ::=  <group_template_header>
                                 "{" <group_template_body> "}"
<group_template_header>     ::=  "group"<identifier>
                                 [ <group_inheritance_spec> ]
<group_inheritance_spec>    ::=  ":" <scoped_name> { "," <scoped_name> }*
<group_template_body>       ::=  [<supporting_def_spec>]
                                 [<interface_def_spec>]
                                 [<object_def_spec>]
                                 [<group_def_spec>]
                                 [<group_predicate_spec>]
                                 <supp_comp_templates>
                                 [<supported_contract_interfaces>]
                                 [<required_contract_interfaces>]
<supporting_def_spec>       ::=  {<supporting_def> ";"}*
<interface_def_spec>        ::=  {<interface_dcl> ";"}*
<object_def_spec>           ::=  {<object_dcl> ";" }
<group_def_spec>            ::=  {<group_dcl> ";" }
<group_predicate_spec>      ::=  "predicate" <string_literal> ";"
<supp_comp_templates>       ::=  "members" <supp_comp> ";"
<supp_comp>                 ::=  <supp_comp_defn> {"," <supp_comp_defn> }*
<supp_comp_defn>            ::=  <scoped_name>
<supported_contract_interfaces  ::=  "supports" <scoped_name>
                                 { "," <scoped_name> }* ";"
<required_contract_interfaces>  ::=  "requires" {<scoped_name> | <tagged_name>}
                                 { "," {<scoped_name> | <tagged_name>} }* ";"
```

A.5.3 Object syntax

```
<object_dcl>                ::=  <object_forward_dcl>
                            |    <object_template>
<object_forward_dcl>        ::=  "CO" <identifier>
<object_template>           ::=  <object_template_header>
                                 "{" <object_template_body> "}"
<object_template_header>    ::=  "CO" <identifier>
                                 [ <object_inheritance_spec> ]
<object_inheritance_spec>   ::=  ":" <scoped_name> { "," <scoped_name> }*
<object_template_body>      ::=  [<supporting_def_spec>]
                                 [<interface_def_spec>]
                                 [<object_behaviour_spec>]
                                 <suptd_interf_templates>
                                 [<reqrd_interf_templates>]
                                 [<object_init_spec>]
<object_behaviour_spec>     ::=  "behaviour" <string_literal> ";"
<reqrd_interf_templates>    ::=  "requires" <req_interf_defn>
                                 {"," <req_interf_defn> }* ";"
<req_interf_defn>           ::=  <scoped_name>
                            |    <tagged_name>
```

```
<suptd_interf_templates>      ::=   "supports" <suptd_interf> ";"
<suptd_interf>                ::=   <suptd_interf_defn> {"," <suptd_interf_defn> }*
<suptd_interf_defn>           ::=   <scoped_name>
<object_init_spec>            ::=   "initial" <scoped_name> ";"
```

A.5.4 Interface syntax

```
<interface_dcl>               ::=   <interface_forward_dcl>
                              |     <interface_template>
<interface_forward_dcl>       ::=   "interface" <identifier>
<interface_template>          ::=   <interface_header> "{" <interface_body> "}"
<interface_header>            ::=   "interface" <identifier>
                                    [ <interf_inheritance_spec> ]
<interf_inheritance_spec>     ::=    ":" <scoped_name> { "," <scoped_name> }*
<interface_body>              ::=   [<supporting_def_spec>]
                                    [<interf_behaviour_spec>]
                                    { <op_sig_defns> | <stream_sig_defns> }
<interf_behaviour_spec>       ::=   "behaviour" {
                                          {<interf_behaviour_text> [<interf_usage_spec>]}
                              |           <interf_usage_spec> }
<interf_behaviour_text>       ::=   "behaviourText" <string_literal> ";"
<interf_usage_spec>           ::=   "usage" <string_literal> ";"
```

A.5.5 (Operational) interface syntax

```
<op_sig_defns>                ::=   { <op_sig_defn> ";" }*
<op_sig_defn>                 ::=   { <announcement> | <interrogation> }
 <announcement>               ::=   "one-way" "void" <identifier> <parameter_dcls>
<interrogation>               ::=   <attr_dcl>
                              |     <oper_dcl>
<attr_dcl>                    ::=   ["readonly"] "attribute" <param_type_spec>
                                    <declarators>
<oper_dcl>                    ::=   <op_type_spec> <identifier>
                                    <parameter_dcls>
                                    [<raises_expr>] [<context_expr>]
<op_type_spec>                ::=   <param_type_spec>
                              |     "void"
<parameter_dcls>              ::=   "(" <param_dcl> { "," <param_dcl> }* ")"
                              |     "(" ")"
<param_dcl>                   ::=   <param_attribute> <param_type_spec>
                                    <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>
                              |     <fixed_pt_type>
                              |     <scoped_name>
```

A.5.6 (Stream) interface syntax

```
<stream_sig_defns>            ::=   { <stream_flow_defn> ";" }*
<stream_flow_defn>            ::=   <flow_direction> <flow_type>
                                    <identifier>
<flow_direction>              ::=   "source" | "sink"
<flow_type>             ::=   <param_type_spec>
```

A.5.7 Supporting definition syntax

```
<supporting_def>        ::=  <const_dcl> ";"
                        |    <type_dcl> ";"
                        |    <except_dcl> ";"
<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>
<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"
<typedcl>               ::=  "typedef" <type_declarator>
                        |    <struct_type>
                        |    <union_type>
                        |    <enum_type>
<type_declarator>       ::=  <type_spec> <declarators>
<type_spec>             ::=  <simple_type_spec>
                        |    <constr_type_spec>
```

```
<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>
<object type>               ::=  "Object"
<floating_pt_type>          ::=   "float"
                            |     "double"
                            |     "long" "double"
<integer_type>              ::=   <signed_int>
                            |     <unsigned_int>
<signed_int>                ::=   <signed_long_int>
                            |     <signed_short_int>
                            |     <signed_longlong_int>
<signed_longlong_int>       ::=   "long" "long"
<signed_long_int>           ::=   "long"
<signed_short_int>          ::=   "short"
<unsigned_int>              ::=   <unsigned_long_int>
                            |     <unsigned_short_int>
                            |     <unsigned_longlong_int>
<unsigned_longlong_int>     ::=   "unsigned" "long" "long"<unsigned_long_int>
                            ::=   "unsigned" "long"
<unsigned_short_int>        ::=   "unsigned" "short"
<char_type>                 ::=   "char"
<wide_char_type>            ::=   "wchar"
<boolean_type>              ::=   "boolean"
<octet_type>                ::=   "octet"
<any_type>                  ::=   "any"
<template_type_spec>        ::=   <sequence_type>
                            |     <string_type>
                            |     <wide_string_type>
                            |     <fixed_pt_type>
<sequence_type>             ::=   "sequence" "<" <simple_type_spec> ","
                                  <positive_int_const> ">"
                            |      "sequence" "<" <simple_type_spec> ">"
<string_type>               ::=   "string" "<" <positive_int_const> ">"
                            |     "string"
<wide_string_type>          ::=   "wstring" "<" <positivi_int_const> ">"
                            |     "wstring"
<fixed_pt_type>             ::=   "fixed" "<" postive_int_const> "," <integer_literal> ">"
<fixed_pt_const_type>       ::=   "fixed"
<constr_type_spec>          ::=   <struct_type>
                            |     <union_type>
                            |     <enum_type>
<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>
<declarators>          ::=  <declarator> { "," <declarator> }*
<declarator>           ::=  <simple_declarator>
                       |    <complex_declarator>
<simple_declarator>    ::=  <identifier>
<complex_declarator>   ::=  <array_declarator>
<array_declarator>     ::=  <identifier> <fixed_array_size>+
<fixed_array_size>     ::=  "[" <positive_int_const> "]"
<positive_int_const>   ::=  <const_exp>
<enum_type>            ::=  "enum" <identifier>
                            "{" <enumerator> { "," <enumerator> }* "}"
<enumerator>           ::=  <identifier>
<except_dcl>           ::=  "exception" <identifier> "{" <member>* "}
```

ANNEX B

Mapping to SDL and ASN.1

B.1 Motivation

As described previously, the ITU-ODL language is intended to be used as a description technique for the static aspects of a computational viewpoint specification. However, for the development of nontrivial systems, a specification of the behaviour of the system could be useful as well. Such a behaviour specification can be done using other languages standardized by the ITU. One of these languages is SDL.

To combine the two languages (ITU-ODL and SDL) a language mapping from ITU-ODL to SDL is proposed in this annex. Within this process all object templates, interface templates, operation signatures and data types are specified in ITU-ODL. In order to specify the behaviour of the application, it is possible to derive a SDL stub from the ITU-ODL specification providing the structures, signatures and data types using the mapping. This stub can be enriched by behaviour by applying the inheritance mechanism. This complete specification allows a testing or simulation of the application before implementation.

Hence, it is needed to have a well-defined language mapping from ITU-ODL to SDL in combination with ASN.1. That language mapping gives the possibility to derive a SDL system from an ITU-ODL specification automatically. The behaviour can be added in a straightforward way just by overloading virtual procedures.

B.2 Basic requirements

Since ITU-ODL is a superset of ODP-IDL, the language mappings contained in [1] could serve as guideline to define a language mapping for ITU-ODL. The following mappings have to be done with respect to the C Language Stub Mapping introduced in [1].

ITU-ODL constructs which also exist in ODP-IDL:

- all ITU-ODL basic data types;
- all ITU-ODL constructed data types;
- constants defined in ITU-ODL;

- references to CORBA-objects[8] defined in ITU-ODL;
- invocation of operations, including passing parameters and receiving results;
- exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed;
- access to attributes.

Additional ITU-ODL specific constructs:

- object templates;
- object group templates;
- behaviour specification;
- stream signatures.

B.3 Structure

An ITU-ODL file is mapped onto two SDL-packages:

- <name>_interface; and
- <name>_definition;

where <name> is the name of the ITU-ODL specification (for instance the file name).

Package <name>_interface contains all information which is relevant for both the client and the server side of the system. In detail these are the data type definitions, constant definitions, definitions of signals for operations, flows, attributes and exceptions and the definitions of signallists. Additionally, there are procedure definitions which can be used by the client to invoke an operation on a server and which handle the exchange of signals between the client and the server (note that this is only a shorthand notation).

Package <name>_definition contains the skeletons for the server side of the system. In fact, there are process types for ITU-ODL interface templates and block types for ITU-ODL object templates and group templates.

B.4 Scoped names

It is required that all templates, types, constants and exceptions must be defined in SDL using their global name. This is required because SDL does not know a scope operator.

If global names are used in an ASN.1 specification, all underscores have to be replaced by hyphens and all type identifiers must start with an upper-case letter. All other identifiers must start with a lower-case letter.

B.5 Module mapping

The ITU-ODL module cannot be mapped in SDL because a scope operator does not exist in SDL. All ITU-ODL constructs like interface templates or data types defined in a module must be visible for all object templates defined outside this module. Therefore, it is needed that all definitions for interfaces templates, data types, object templates, etc. included in a module must be done in the SDL packages using their global name (cf. B.4).

---

[8] There is a clash in the definition of the term "object" in OMG and ODP. This Recommendation relies on the ODP definition. If the OMG definition is referred, it is noted as CORBA-object.

B.6 Interface template, operation, flow and attribute mapping

ITU-ODL interface templates are mapped onto process types contained in the package <name>_definition. These process types contain the whole interface template behaviour specification as a comment (informal text), the operations provided at that interface as virtual procedures, the flows as remote variable declarations and the attributes as local variable declarations.

Operations

The operations cannot be mapped to remote procedures currently. This is due to the fact that SDL does not contain a mechanism for handling exceptions. However, the exception concept is a fundamental requirement when describing distributed systems. Therefore, operations are mapped to a set of signals as follows:

- pCALL_<global name of operation>;
- pREPLY_<global name of operation> (not for one-way operations).

The pCALL signal carries all in and inout parameters of the operation; the pREPLY signal carries all out and inout parameters of the operation and the return value. The signals are defined in the package <name>_interface. The process type generated for the interface template in package <name>_definition contains a local procedure for the operation. The name of the procedure is the operation name. The parameters of the operation are declared as in or in/out parameters in the SDL procedure. If an parameter is specified as out in ITU-ODL, it is mapped onto an in/out parameter in SDL.

A client can invoke an operation at a server by sending the proper pCALL signal. The server receives the signal and (according to its state) it calls the local procedure which contains the implementation of the operation. The result is passed back to the client using the pREPLY signal.

NOTE 1 – To simplify the call mechanism on the client side, the package <name>_definition contains a procedure for each operation which sends the pCALL signal, waits for the answer in wait state and returns the result to the client. See the example below.

NOTE 2 – Exceptions are also mapped to signals, and raising an exception is nothing more than sending the appropriate exception signal back to the client instead of the pREPLY signal.

NOTE 3 – To make sure that pCALL and pREPLY signals are not mixed up, each invocation has to contain a unique identifier, which is afterwards included in the termination. This mapping does not prescribe a concrete way to implement this.

```
exception ex{
};

interface i{
        behaviour
                behaviourText
                  "The interface serves for contacting type management";
                usage
                  "Operation op must be invoked prior to other operations defined on the service ";

        string op (
                in boolean p1,
                inout char p2,
                out octet p3
        )
         raises ( ex);

        one-way void op1(
                in double p1
        );
} /*end opr interface */;
```

```
use idltypes ;
package name_interface ;
     /* name shall be replaced by the name of the ODL specification */
     signal pCALL_i_op(ODL_boolean,ODL_char);
     signal pREPLY_i_op(ODL_char,ODL_octet,ODL_string);
     signal pCALL_i_op1(ODL_double);
     signal pREPLY_i_op1;
     signal pRAISE_ex;
     signallist i_INVOCATIONS=pCALL_i_op,pCALL_i_op1 ;
     signallist i_TERMINATIONS=pREPLY_i_op,pRAISE_ex ;
     procedure i_op ;
          fpar in p1 ODL_boolean,
               in/out p2 ODL_char,
               in/out p3 ODL_octet,
               in server ODL_Object ;
          returns ODL_string ;
          dcl ex_var ex,return_var ODL_string ;
          start;
               decision (server);
                    (Null): output pCall_i_op(p1,p2);
                    else: output pCall_i_op(p1,p2) to server ;
               enddecision;
               nextstate Wait ;
          state Wait ;
               save * ;
               input pReply_i_op(p2,p3,return_var);
                    output pReply_i_op(p2,p3,return_var) to self ;
                    return return_var;
               input pRAISE_ex;
                    output pRAISE_ex to self;
               return return_var;
          endstate Wait ;
     endprocedure i_op ;
endpackage name_interface ;

use idltypes;
use name_interface ;
package name_definition ;
     /* name shall be replaced by the name of the ODL specification */
     process type <<package name_definition>> i /*interface definition i*/
          comment
               "The interface serves for contacting type management
               Operation op must be invoked prior to other operations defined on
               the service ";
          gate i_INVOCATIONS in with (i_INVOCATIONS) ;
          gate i_TERMINATIONS out with (i_TERMINATIONS) ;
          dcl  op_p1 ODL_boolean,op_p2 ODL_char,op_p3 ODL_octet,op_return
          ODL_string,op1_p1 ODL_double ;
          virtual procedure <<package name_definition/process type i>> op ;
               fpar in p1 ODL_boolean,
                    in/out p2 ODL_char,
                    in/out p3 ODL_octet ;
               returns ODL_string ;
          endprocedure <<package name_definition/process type i>> op ;
          virtual procedure <<package name_definition/process type i>> op1 ;
               fpar in p1 ODL_double ;
          endprocedure <<package name_definition/process type i>> op1 ;
     endprocess type <<package name_definition>> i ;
endpackage name_definition ;
```

Flows

Stream signatures are represented in SDL as remote variable declarations. The producer exports the stream variable and the consumer imports it. That means on the server side a flow specified as sink is mapped onto an imported variable specification and a flow specified as source is mapped onto an exported variable declaration. These variable declarations are contained in the process type generated for the interface template using their global name. On client side it has to be handled vice versa, but this is not generated automatically.

NOTE – Note that the imported and exported variables have to be declared as remote in package <name>_definition.

References

ITU-ODL interface references will be denoted as SDL PIds. These PIds can be arguments, return results or components of type definitions.

For all interface templates a new type is introduced representing the reference to that interface.

Example:

```
interface i { /* ITU-ODL */
...
};
IRef ::= PId; /* SDL + ASN.1 */
```

A call from a client to server addressed by a special reference is performed by calling the procedure contained in package <name>_interface and adding the PId of the server as the last parameter.

Attributes

ITU-ODL attribute declarations defined in an interface template are mapped onto local variable definition within the process type definition representing the ITU-ODL interface template. Additionally two remote procedures exported by the process type are defined allowing the remote access (read and write) to these attributes by the client which imports these remote procedures. In case an attribute is declared readonly within the ITU-ODL specification only one remote procedure is provided allowing to read the values of the corresponding attribute. The remote declaration is done in package <name>_interface.

Example:

```
interface i3 {       /* ITU-ODL */
  attribute boolean attr1;
  readonly attribute short attr2;
};
use idltypes ;
package name_interface ;
     /* name shall be replaced by the name of the ODL specification */
     remote procedure i3__set_attr1;
          fpar in ODL_boolean;
     remote procedure i3__get_attr1;
          returns ODL_boolean;
     remote procedure i3__get_attr2;
          returns ODL_short;
endpackage;
use idltypes;
use name_interface;
package name_definition;
     /* name shall be replaced by the name of the ODL specification */
```

```
    process type i3;   /* SDL + ASN.1 */
         dcl
              attr1 ODL_boolean,
              attr2 ODL_short;
         exported procedure i3__set_attr1 referenced;
         exported procedure i3__get_attr1 referenced;
         exported procedure i3__get_attr2 referenced;
    endprocess type i3;
    exported procedure i3__set_attr1;
         fpar
              in value ODL_boolean;
         start;
              task
          attr1 := value;
         return;
    endprocedure i3__set_attr1;
    exported procedure i3__get_attr1;
         returns ODL_boolean;
         start;
              return attr1;
    endprocedure i3__get_attr1;
    exported procedure i3__get_attr2;
         returns ODL_short;
         start;
              return attr2;
    endprocedure i3__get_attr2;
endpackage;
```

B.7 Interface template inheritance

The inheritance of interface templates defined in ITU-ODL has to be represented as a flat structure. That means all constructs for operations, flows and attributes defined in the base interface template have to be defined in the derived interface template again. The behaviour specifications are not inherited.

The inheritance mechanism of SDL cannot be used because only single inheritance is supported there and not multiple inheritance like in ITU-ODL.

If the specification does only contain single inheritance, the SDL inheritance feature can be applied[9].

B.8 Mapping for object templates

Object templates are represented in SDL as block types defined in package <name>_definition using the global name of the object template. Similar to interface templates, the behaviour specification of object templates is mapped onto a comment. The other parts of the object template are mapped as follows:

Required interfaces

The list of required interface templates is mapped to a comment.

Supported interfaces

The interface templates announced in the list of supported interfaces are mapped in the following way:

[9] It is subject of the used tools to decide whether to use SDL inheritance or not.

For each interface template a new virtual process types is introduced in the block type representing the object template. These process types inherit the corresponding process type defined for the interface template (see B.6). It has the global name of the interface template.

Initialization specification

For the interface template announced in the initial construct, a new virtual process type is introduced in the block type definition. These process types inherit the corresponding process types defined at system type level (see B.6). It has the global name of the interface template.

Inheritance

Object template inheritance has to be mapped as a flat structure. This is due to the fact that SDL does not have the feature of multiple inheritance. If only single inheritance is contained in the ITU-ODL file, the SDL inheritance can be used.

Example:

```
interface i;          /*ITU-ODL*/
interface i1;
interface i2;
CO o {
      behaviour
            "use i to initialize object" ;
      requires
            i2;
      supports
            i1;
      initial
            i;
} /*end object */;

use idltypes ;
package name_interface ;
...
endpackage name_interface ;

use idltypes ;
use name_interface ;
package name_definition ;
     /* name shall be replaced by the name of the ODL specification */
     process type <<package name_definition>> i /*interface definition i*/;
          ...
     endprocess type <<package name_definition>> i ;
     process type <<package name_definition>> i1 /*interface definition i1*/;
          ...
     endprocess type <<package name_definition>> i1 ;
     process type <<package name_definition>> i2 /*interface definition i2*/;
          ...
     endprocess type <<package name_definition>> i2 ;
     block type <<package name_definition>> o /*object definition o*/
          comment "use i to initialize object"
          comment "requires i2";
          process type <<Block type o>> i1 /*supported interface i1*/
               inherits <<package name_definition>> i1;
          endprocess type <<Block type o>> i1;
          process type <<Block type o>> i /*initial interface i*/
               inherits <<package name_definition>> i;
          endprocess type <<Block type o>> i;
```

```
     endblock type <<package name_definition>> o ;
endpackage name_definition ;
```

B.9 Mapping for object group templates

Object group templates are mapped onto block types in package <name>_definition using the global name of the group template. These block types contain a block substructure with block type definitions for each of the members of the group template inheriting form the block types generated for the members itself. It has the global name of the members template.

The group template predicate is mapped onto a comment.

The contracts are not mapped.

Group template inheritance has to be mapped as a flat structure. This is due to the fact, that SDL does not have the feature of multiple inheritance. If only single inheritance is contained in the ITU-ODL file, the SDL inheritance can be used.

Example:

```
interface i;          /*ITU-ODL*/
interface i1;
interface i2;

CO o;

group g{
     predicate "All group members have the following security policy:..." ;
     members ::o;
} /*end group */;

use idltypes ;
package name_interface ;
...
endpackage name_interface ;

use idltypes ;
use name_interface ;
package name_definition ;
     /* name shall be replaced by the name of the ODL specification */
     ...
     block type <<package name_definition>> o /*object definition o*/ ;
     ...
     endblock type <<package name_definition>> o ;

     block type <<package name_definition>> g /*group definition g*/;
          block type <<Block type g>> o /*group member definition o*/
               inherits <<package name_definition>> o;
          endblock type <<Block type g>> o;
     endblock type <<package name_definition>> g ;
endpackage name_definition ;
```

B.10 Mapping for constants

Constants defined in ITU-ODL are mapped to synonyms in SDL.

Example:

```
const float x = 7.5 + 3.4;        /* ITU-ODL */

synonym x ODL_REAL = 7.5 + 3.4;   /* SDL + ASN.1 */
```

B.11 Mapping for basic data types

In this subclause it is shown how ITU-ODL basic data types are expressed in ASN.1. All basic data types are defined as ASN.1 types as shown in the table below[10]. Each type gets a name like the following: ODL_<typename> where type name is one of the ITU-ODL basic data types. Variables of these types can be declared in a SDL dcl statement and can be used in the same way as variables of normal SDL data types. The way how to use the ASN.1 data types in a SDL specification is described in [7].

ITU-ODL basic data type	Corresponding ASN.1 data types
Short	**INTEGER(-2**^{15}**..2**^{15}-1)
Unsigned short	**INTEGER(0..2**^{16}-1)
Long	**INTEGER(-2**^{31}**..2**^{31}-1)
Unsigned long	**INTEGER(0..2**^{32}-1)
Float	**REAL** (represents IEEE single-precision floating point numbers[11])
Double	**REAL** (represents IEEE double-precision floating point numbers)
Char	**VisibleString SIZE(1)**
Boolean	**BOOLEAN**
Octet	**BIT STRING SIZE(8)**
Any	**ANY**

[10] As an alternative, the data types can be mapped to SDL data types. This mapping is not contained in this annex.

[11] IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Std. 754-1985.

B.12 Mapping for constructed data types

B.12.1 Mapping for structure types

Structured types defined in ITU-ODL as struct are represented in ASN.1 as a SEQUENCE. Members of the same type in structure can be noted as a list as in ITU-ODL. That is not possible in an ASN.1 SEQUENCE. Therefore, every member must be specified separately.

Example:

```
struct S {    /* ITU-ODL */
  long mem1;
  short mem2;
};
S ::= SEQUENCE {    /* SDL + ASN.1 */
   mem1 ODL-long,
   mem2 ODL-short
};
```

There are operators to generate an instance of the type defined above and to access and to modify its members. These operators are explained in [7].

```
Dcl
     s S,
     i ODL_long,
     j ODL_short;
...
     task
          s := (. 2, 1 .),
          i := s!mem1,
          j := s!mem2,
          s!mem2 := 5;
```

B.12.2 Mapping for union

An ITU-ODL union is mapped on an ASN.1 SEQUENCE containing a tag indicating which type is meant and a CHOICE of all types defined in the ITU-ODL union.

Example:

```
union u switch (short) { /* ITU-ODL */
  case 1: long l;
  case 2: short s;
  default: float f;
};
```

```
U ::= SEQUENCE {    /* SDL + ASN.1 */
   tag ODL_short,
   union CHOICE {
      l ODL-long,
      s ODL-short,
      f ODL-float
   }
};
```

The discriminator is always a named tag and the CHOICE is a named union.

```
dcl
   u U,
   ll ODL_long,
   ss ODL_short,
   ff ODL_float;
...
/* making a call to get a value for u */
...
decision (u!tag);
   (1): task
            ll := u!union!l;
   (2): task
            ss := u!union!s;
   else: task
            ff := u!union!f;
enddecision;
```

B.12.3 Mapping for enumeration

ITU-ODL enumerations (enum) are represented in ASN.1 as ENUMERATED. The ordering of the enumeration elements in must not be changed. An explicit order in the ASN.1 enumeration has to be outlined.

For enumeration types there are relational operators and operators for getting the first and last enumerator and the predecessor and successor of each enumerator. These operators are given in [7].

B.12.4 Mapping for sequence types

An ITU-ODL SEQUENCE is mapped onto an ASN.1 SEQUENCE OF type, either with size boundary or not, depending on the ITU-ODL SEQUENCE description.

Example:

```
typedef sequence <short, 30> seq;  /* ITU-ODL */
SEQ ::= SEQUENCE SIZE(30) OF ODL-short;    /*SDL + ASN.1 */
```

The operators to access and to modify the members of the SEQUENCE OF are defined in [7].

B.12.5 Mapping for strings

An ITU-ODL STRING is represented in ASN.1 as aVisibleString, either with size boundary or not, depending on the ITU-ODL STRING description.

The ASN.1 type gets the name ODL_string.

B.12.6 Mapping for arrays

An ITU-ODL array is mapped onto an ASN.1 SEQUENCE OF type for every dimension defined with size boundary.

Example:

```
<type> a[8][7][67];       /* ITU-ODL */
```

If it is a member of a struct:

```
a SEQUENCE SIZE(8) OF SEQUENCE SIZE(7) OF SEQUENCE
                                SIZE(67) OF <type>;
```

If it is a type specification:

```
A ::= SEQUENCE SIZE(8) OF SEQUENCE SIZE(7) OF SEQUENCE
                                       SIZE(67) OF <type>;
```

with respect to whether the array is a member of a struct or a type specification.

B.13 Mapping for exceptions

For every EXCEPTION defined in an ITU-ODL specification, an ASN.1 SEQUENCE type is defined in the package <name>_interface with the global name of the exception. This SEQUENCE contains the parameters of the exception. Additionally, there is a signal definition pRAISE_<name> where name is the global name of the exception. This signal carries the data type defined for the exception parameters. The server which wants to raise an exception simply sends the corresponding pRAISE signal to the client. The client can then obtain the values of the exception parameters from the signal.

Example:

```
exception ex{                    /*ITU-ODL*/
     long p1;
};

use idltypes ;                   /*SDL,ASN.1*/
package name_interface ;
     /* name shall be replaced by the name of the ODL specification */
     Ex ::= SEQUENCE {
          ODL_long
     };
     signal pRAISE_ex( Ex );
endpackage;
```

B.14 Additional definitions

There are some additional definitions proposed here which are not necessary but intended to make the handling of the generated SDL specification easier.

Signallists

For each operational interface template a signallist is defined in package <name>_interface containing the (pCALL) signals for the operations which can be invoked on that interface. A second signallist contains all possible terminations: that means all pREPLY and pRAISE signals for the possible terminations of the operations. The two signallists are named <name>_INVOCATIONS and <name>_TERMINATIONS, where <name> is the global name of the interface template.

Gates

The process types generated for operational interface templates contain two gate definitions: an outgoing gate with the TERMINATIONS signallist and an incoming gate with the INVOCATIONS signallist. The gates are named <name>_INVOCATIONS and <name>_TERMINATIONS, where <name> stands for the name of the interface template.

ANNEX C

Mapping to C++

C.1 Motivation

ITU-ODL is a strict superset of ODP-IDL [8]. All aspects concerning the communication between the components of a system are described using ODP-IDL interfaces descriptions. In order to be able to use existing CORBA products as a platform to implement an ITU-ODL specification, the mapping for the ODP-IDL part is adopted from OMG [1]. The intention is that the existing ORB specific ODP-IDL compilers can be used to map the ODP-IDL part of an ITU-ODL specification to C++.

Additionally, the component or component information should be reflected in the implementation language as well. This information can be considered as design information. It is not a requirement to map it in the implementation language. The application will work even if only the ODP-IDL part is mapped (only the interfaces are of importance for communication). But ITU-ODL should be understood as a computational design language and the information contained in it should make the programming of distributed applications easier. Therefore, a language mapping of ITU-ODL to C++ [10] is proposed in this Recommendation.

C.2 Basic requirements

Since ITU-ODL is a superset of ODP-IDL, the language mappings contained in [1] serve as guideline to define a language mapping for ITU-ODL. The following concepts have to be mapped with respect to the C++ Language Stub Mapping introduced there.

ITU-ODL constructs which also exist in ODP-IDL:

- all ITU-ODL basic data types;
- all ITU-ODL constructed data types;
- constants defined in ITU-ODL;
- references to interfaces defined in ITU-ODL;
- invocation of operations, including passing parameters and receiving results;
- exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed;
- access to attributes.

Additional ITU-ODL specific constructs:

- object templates (COs);
- group templates;
- behaviour specification;
- stream signatures (there is currently no stream mapping defined in this document).

C.3 Structure

An ITU-ODL file is mapped onto C++ header and (possibly) implementation files. There is no prescribed structure of the generated files.

C.4 Scoped names

The rules for global names in ITU-ODL are the same as in ODP-IDL. There are only three more entities forming a namescope: stream interface templates, CO templates and group templates. All entities can be accessed in C++ using the scope operator "::". No additional rules are needed.

C.5 Module mapping

Modules are supposed to be mapped on C++ namespace. Since a namespace can be reopened, there is no problem using an ORB specific ODP-IDL compiler and generate additional files for the group and object templates of ITU-ODL.

If due to compiler or ORB restrictions a mapping on namespaces is not possible, the reopening problem must be solved otherwise. This could be done by locating the group and object template definitions in an additional class ::ODL. In that case, global names referring to COs or groups must be prefixed with that class name.

C.6 Interface template, operation, flow and attribute mapping

Operational interface templates, operations and attributes are mapped onto C++ classes in the same way as described in the ODP-IDL to C++ mapping [1].

C.6.1 Behaviour and usage clauses

The behaviour descriptions and usage descriptions are plain text only. They will be mapped onto comments. If an existing ODP-IDL compiler is used, they are not mapped.

C.6.2 Flows

Currently not available.

C.6.3 Interface template inheritance

Interface template inheritance is mapped in the same way as described for the ODP-IDL to C++ mapping.

C.7 Mapping for object templates

Object templates are mapped to C++ classes (CO classes). The main functionality which is provided by the generated classes is that they contain methods for creation and deletion of the supported interfaces of the ITU-ODL CO definition.

Additional functionality such as checkpointing or migration can be provided but is not prescribed and depends on the platform on which the application should run. For that reason it might be useful to have a common base class similar to ::CORBA::Object, and let all generated classes inherit from it.

C.7.1 Required interface specification

Required interface templates are not mapped themselves. There is the CORBA client mapping for the interface template only.

However, the required interface information is needed if a minimal server implementation (with respect to the count of generated classes) should be built. In that case, only the client part of the CORBA mapping for those interfaces which are required and the server part for the interfaces which are supported has to be generated.

C.7.2 Supported interface specification

The interface templates announced in the list of supported interfaces on an arbitrary CO are mapped in the following way:

- There is a C++ class generated which implements the class generated according to the ODP-IDL mapping. This class contains implementations for all operations of the interface. The behaviour is to delegate an incoming call to an implementation class. The programmer does not have to touch any of these delegation classes.
- The implementation class is generated for all interfaces listed somewhere as supported. The implementation class contains all operations as pure virtual methods. The programmer is responsible to provide implementations of these implementation classes.
- There is a possibility to make a reference to an instance of an implementation class known to the delegation class instance. This can be implemented either in the constructor of the delegation class or by a method _ _set_implementation.

The CO can create instances of its supported interface templates in the following way:

1) Create an instance of the delegation class.

2) Obtain a reference to an instance of the implementation class (can exist or be created).

3) Make the reference to the implementation class instance known to the delegation class instance.

C.7.3 Initialization specification

The initial interface is mapped in the same sense as supported interfaces.

C.7.4 Inheritance

Object template inheritance is realized as C++ inheritance between the generated C++ classes.

C.7.5 Example

Suppose that the following ITU-ODL definition has to be mapped to C++.

```
interface i3;        /*ITU-ODL*/
interface i1;
interface i2;
CO o {
        behaviour
                "use i to initialize object" ;
        requires
                i1;
        supports
                i2;
        initial
                i3;
} /*end object */;
```

Figure 7 shows the generated classes and their relations. The arrows represent the inheritance relation whereas the dashed arrows symbolize delegation. The classes suffixed with _impl are generated by most ODP-IDL compilers for the implementation of interfaces. The classes suffixed with _dpe are the delegation classes and the classes suffixed with _IMPL are the implementation classes introduced above. The class dpe is a base class for all delegation classes and the class dpe_o is a base class for all CO classes. They are not prescribed by this mapping.

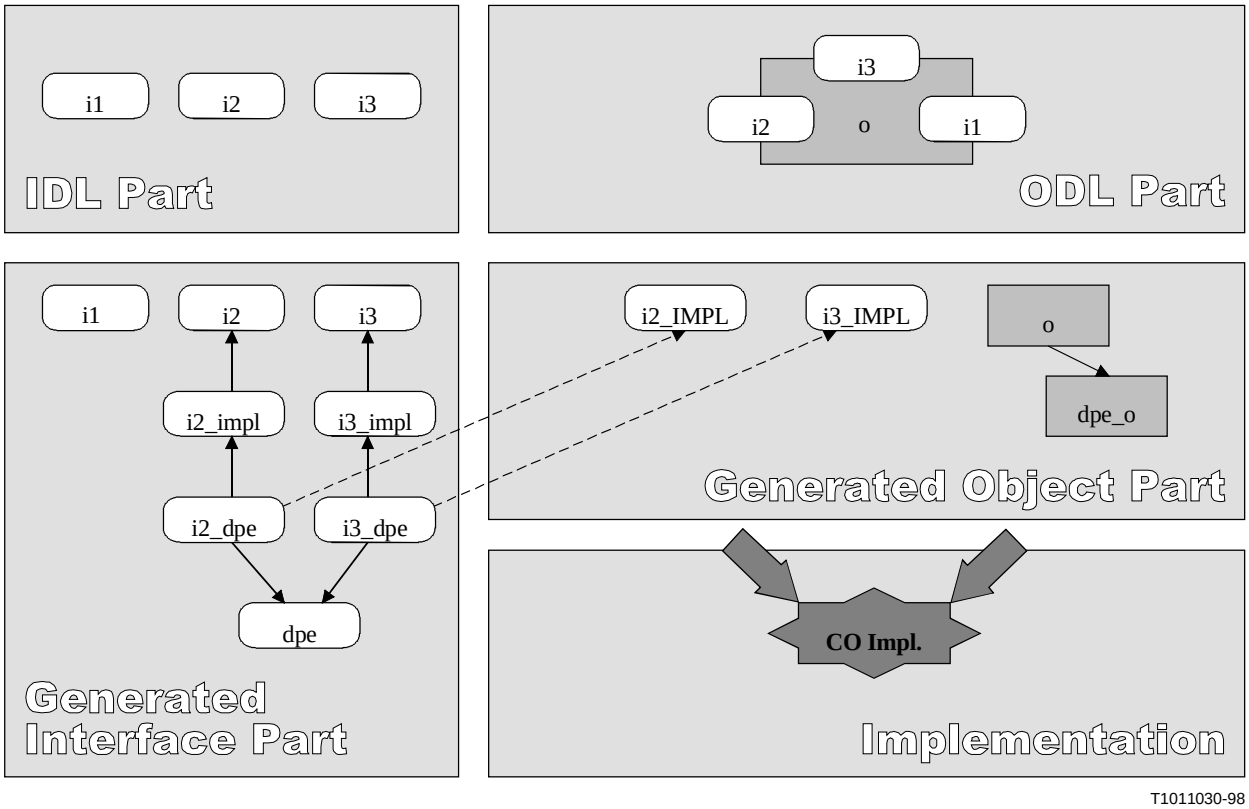

Figure 7/Z.130 – Mapping of object templates and supported interface templates

The programmer is free in the choice how to structure the application. He must only provide implementations for the implementation classes and possibly for the CO classes.

Figure 8 shows some examples how the implementation might be structured:

- There is the possibility to have one implementation for each interface implementation class. That normally happens when there is a lot of state information related to the interface. The synchronization and information exchange is handled via the CO class implementation.
- It is possible to have one implementation for a number of interface implementation classes.
- If there is no state information related to the interfaces, the implementation could be done directly in the CO class implementation.

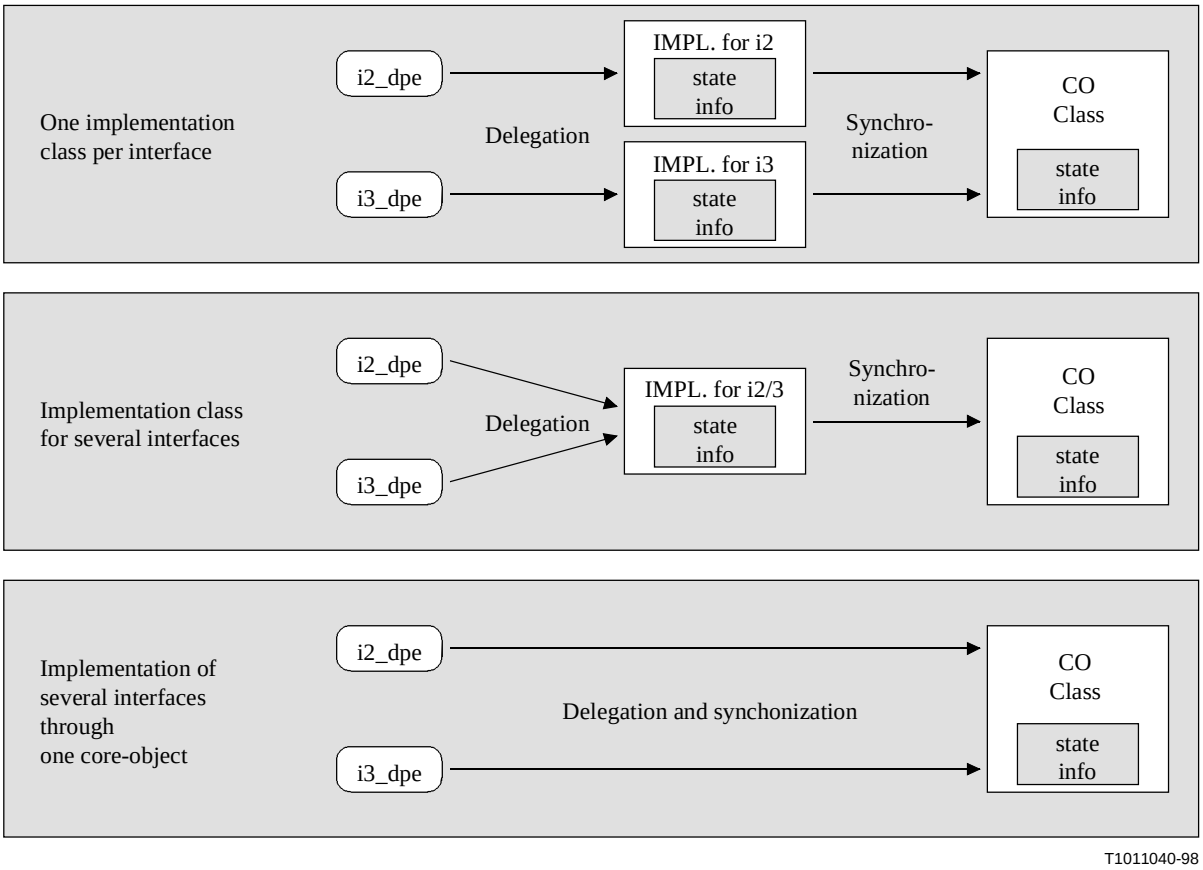

Figure 8/Z.130 – Different implementation strategies

C.8 Mapping for group templates

Group templates are sets of COs and computational groups which can be clustered together for any reason. The reason for grouping them is contained in the group predicate specification. Since the predicate specification can vary, there is no prescribed mapping for groups.

However, if the purpose of the group is to group its members for implementation, the group can be mapped onto a class definition in the same way as COs. The contracts of the group (the supported and required interfaces) are then mapped in the same way as for objects.

C.9 Mapping for constants

Constants are mapped in the same way as described in the ODP-IDL to C++ mapping.

C.10 Mapping for basic data types

Basic data types are mapped in the same way as described in the ODP-IDL to C++ mapping.

C.11 Mapping for constructed data types

Constructed data types are mapped in the same way as described in the ODP-IDL to C++ mapping.

C.12 Mapping for exceptions

Exceptions are mapped in the same way as described in the ODP-IDL to C++ mapping.

APPENDIX I

Quality of service

I.1 Motivation

Specification of the functional aspects of an operation, or flow, may need to be augmented with a specification of the "standard of the service" required. There are a number of ways one might want to state information about such quality of service. For example, there may be:

- Statements of mandatory capabilities. For example, a flow must support connections of a certain bandwidth (no more and no less); otherwise, it is not offered.
- Statements of expectation. For example, an operation must respond within a certain time for most cases.
- Statements of support. For example, when binding to a stream interface instance, the quality of service is to be negotiated using say, bandwidth and jitter.

With regard to what is "specifically the subject of quality of service information", only a vague definition is offered here. The qualities of service associated with an operation or flow are the timing constraints, degree of parallelism, availability guarantees and so forth, to be provided by an object. Quality of service information deals with the provision of the service, rather than the service itself. Quality of service specifications allow statements about the "level of service" offered by constructs. In general this information may be provided at specification time, or dynamically by the management system responsible for initiating object creation. It may even be altered during the lifetime of the service offering.

The problem of adding quality of service specifications to ITU-ODL can be seen more as a problem of semantics than syntax. This is highlighted when one considers that for a given syntax it is likely that all three of the above interpretations might be possible when one guesses at the meaning of a simple syntax. For example, a quality of service statement associated with a flow of BandwidthType bandwidth = 3 Mbit/s may mean that the interface with that flow cannot be offered unless it can source/sink exactly 3 Mbit/s, or typically connections to the flow are made at 3 Mbit/s; but other values may be negotiated, or the maximum bandwidth of a connection is at 3 Mbit/s, and only values lower may be negotiated, and so forth. The particular quality of service semantics to be supported in ITU-ODL has not been finalized.

In this version of ITU-ODL, semantics is left to the programmer.

NOTE – The ongoing work within ITU-T on Open Distributed Processing – Reference Model – Quality of Service" should be integrated after approval.

I.2 Syntax

ITU-ODL allows to associate a quality of service type and variable identifier to any definition of:

- operation; or
- flow.

Values are not ascribed to the quality of service variables at specification time. Instead they are to be assigned at instantiation time. The semantics are service dependant. This capability is acknowledged as being severely restricted. The semantic of a quality of service specification can be described in the interface behaviour specification.

Following is the syntax of an operational quality of service specification. Note that the quality of service parameters are added after the keyword "with". Note that the identifier of the defined variable must be unique within the interface:

```
<op_sig_defn>          ::=   { <announcement> | <interrogation> }
                             ["with" <QoS_attribute>]
<QoS_attribute>        ::=   <QoS_attr_type> <QoS_attr_name>
<QoS_attr_type>        ::=   <simple_type_spec>
<QoS_attr_name>        ::=   <simple_declarator>
```

The syntax for attaching a quality of service specification to a flow is as follows:

```
<stream_flow_defn>     ::=   <flow_direction> <flow_type>
                             <identifier> ["with" <QoS_attribute>]
```

I.3 Example

Following is an example of a flow quality of service specification. The quality of service parameters appear after the keyword "with". Quality of service is offered using an instance of VideoQoS. An instance of VideoQoS is represented as a float, but depending upon the value of Guarantee (either Statistical or Deterministic), the float is to be interpreted as a "mean" or "peak" frame rate.

```
struct VideoQoS {
        union Throughput switch (Guarantee) {          /* in frames/s */
              case Statistical:        float   mean;
              case Deterministic:      float   peak;
        };
}; // end of VideoQoS

interface I3 {
        ...
        sink VideoFlowType display with VideoQoS requiredQoS;
        ...
}; // end of I3
```

ITU-ODL does not specify standard quality of service parameters for operations or flows.

I.4 Mapping to SDL

The mapping for a quality of service specification to SDL is similar to the mapping of readonly interface attributes. For each quality of service parameter there is a local variable in the process type generated for the server side together with a get procedure to obtain its value. The naming conventions are the same as described in B.6.

APPENDIX II

Comparison of ITU-ODL with ODP-IDL and TINA-ODL

II.1 ITU-ODL objective vs. ODP-IDL objective

The objectives of ITU-ODL are the following:

- language for application specification (at development time);
- language for application re-use (at development time);
- language supporting application execution and interaction (at run-time).

ODP-IDL shares most of these objectives (support for application specification, application re-use, and application interaction).

II.2 Object model

The object model which is supported by ITU-ODL extends the OMG Object Model [1] in the following ways:

At the object level, ITU-ODL offers support for the definition of:

- object behaviour;
- objects with multiple interfaces;
- object groups.

At the interface level, ITU-ODL offers support for the definition of:

- stream interfaces;
- interface behaviour.

At the operation and flow level, ITU-ODL offers support for the definition of:

- stream (flow) signatures.

II.3 ITU-ODL syntax vs. ODP-IDL syntax

ITU-ODL in its current version is a superset of ODP-IDL. It implies that it is possible to use ODP-IDL specifications as part of ITU-ODL specifications (as operational interface declaration). Consequently, the syntax defined in ITU-ODL for operational interface declaration encompasses and supports all the rules defined for ODP-IDL [8].

II.3.1 General syntax

For an ITU-ODL specification, broadly:

- the structures added to ODP-IDL are the object declarations (<object_dcl>), and the object group declarations (<group_dcl>);
- the structures shared with ODP-IDL are the supporting definition (<supporting_def>), and the module declaration (<module>);
- the structure modified from ODP-IDL is the interface declaration (<interface_dcl>).

II.3.2 Interface syntax

In the syntax for interface declaration:

- the structures added to ODP-IDL are the (optional) declaration of interface behaviour (<interf_behaviour_spec>), interface usage specification (<interf_usage_spec>) and the stream (flow) signature declaration (<stream_sig_defns>);
- the structures shared with ODP-IDL are the forward declaration of interfaces (<interface_forward_dcl>), the interface header keyword and name ("interface" and <identifier>), and the interface inheritance specification (<interf_inheritance_spec>);
- the structure modified from ODP-IDL is the operation signature declaration (<operation_sig_defns>).

II.3.3 Operation syntax

In the syntax for operation declaration:

- the structures shared with ODP-IDL are the announcement declaration (<announcement>), and the interrogation declaration (<interrogation>).

Note that all the additions to the ODP-IDL have been made optional.

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the 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	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
Series Z	**Languages and general software aspects for telecommunication systems**