Telecommunications
Information
Networking
Architecture
Consortium

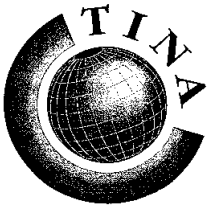
TINA-C Deliverable

Issue Status: Publicly Released

TINA Object Definition Language Manual

Version: 2.3

Date of Issue: 22 Julyr 1996



Telecommunications
Information
Networking
Architecture
Consortium

TINA-C Stream Deliverable

Issue Status: Stream Draft

TINA Object Definition Language MANUAL

Version: 2.3

Date of Issue: July 22 1996

Abstract: This document presents the TINA Object Definition Language (ODL). ODL is a language used to specify applications in the computational viewpoint of TINA. It is a superset of OMG IDL. ODL enables the specification of computational objects comprising *operational* and *stream* interfaces, as well as the specification of agregations of computational objects into *object groups*. Additional features of the language include inheritance, quality of service specification and interface trading support.

Editor: A. Parhar

Stream: DPE

Workplan Task: MTO1.2, 189.1, 190.1, 190.2,194.1, 198.1, 199

File Location: /u/tinac/96/dpe/viewable/odl_manual_v2.3.ps

Archiving Label: TR_NM.002_2.2_96

Extended Abstract

This document specifies the syntax of the TINA Object Definition Language (ODL). ODL is used for the specification of TINA systems from the perspective of the *TINA computational viewpoint*. It defines templates for *operational interfaces*, *stream interfaces*, *multiple-interface objects*, and *object groups*. Version 2.3 of ODL, as presented in this document, is sometimes referred to as ODL capability set 2, and supersedes version 1.3, which is also referred to as ODL capability set 1.

ODL is an extension of the Object Management Group's (OMG) Interface Definition Language (IDL), with additions to support the specification of TINA architectural components. ODL is intended to be a superset of OMG-IDL. This relationship between ODL and OMG-IDL supports the construction of TINA systems over OMG specified Object Request Broker (ORB) implementations.

This document should be read and used by those who deal with TINA computational specifications. This will include designers of TINA based software systems and the designers of supporting CASE tools. The rules and definitions comprising ODL should be followed to achieve TINA compliant computational specifications. Similarly, the rules and definitions of ODL can be embodied in CASE tools supporting software development based on such specifications.

The readers of this document are expected to be familiar with the TINA computational modelling concepts [5], including object grouping concepts [23]. Familiarity with OMG-IDL [37] would be helpful.

Changes from Last Version

Major Changes Made from Previous Version

ODL version 2.1 is defined in this document. It has evolved over time, and is founded upon concepts outlined in previous versions of the document.

Version History

Version1.0 Initial Draft

Version 1.1 Released within the DPE stream for comment, and to some member companies for initial comment.

Version 1.2 Contained many grammatical and other corrections to Version 1.1. Most changes between Version 1.1 and Version 1.2 were minor, although some sections were expanded for clarification. These changes were based on internal stream reviews.

This version was released for wide review amongst member companies and the core team.

Version 1.3 Contains changes made as a result of Core Team and member company review.

Updated to conform to new TINA-C Core Team document format.

Restructured to more clearly define Capability Sets for ODL. Versions 1.x referring to Capability set 1, 2.x to Capability Set 2, and so forth

Changed the syntax for initialization of objects and removed the syntax for usage specification at the object level.

Removed all syntax dependent on transactions from CS-1. Transaction support moved to future work.

Changed syntax for Quality of Service attributes in interfaces to apply more explicitly to operations or flows in the interface.

Changed syntax for Quality of Service constraints in object templates.

Included description of stream interface syntax, and updated section on inheritance.

Version 2.0 Initial Draft of Capability Set 2, ODL specification.

Allows object inheritance, grouping and specification of interface attributes.

Version 2.1 Revised version of Capability Set 2

ODL syntax reorganized. Mainly limited to refinement of additions in Version 2.0.

Document structure changed. Original structure presented, and related, concepts as interfaces and objects to which is added inheritance, object groups, and additional features. The new structure presents, and relates, interfaces, objects and *object groups*, to which is added inheritance, quality of service and other features.

These two changes are driven by the increasing size of ODL syntax, and a need to better document the semantics associated with the syntax.

Version 2.2 Capability Set 2 revised to take into consideration stream review comments. The major changes are as follows:

- quality of service for interfaces, objects and groups removed;
- reference made to quality of service architecture previously specified in TINA-C;
- inheritance rules made more coherent and comprehensive;
- syntactic order of specifications changed to simplify the construction of parsers, required interfaces of objects made part of object behavior; behavior for interfaces, objects and groups given structure;
- future work section expanded to include major unresolved issues.

Table of Contents

1. Introduction

TINA Object Definition Language (ODL) has been developed in the TINA Core Team to:

- Document TINA computational specifications. For example, a TINA entity, such as a User Agent, can be described from the computational viewpoint using an ODL specification.
- Provide syntax suitable for developing software engineering support applications such as ODL parsers, computational specification editors and related CASE tools.

ODL is an extension of the Object Management Group's Interface Definition Language (OMG-IDL) [37]. ODL supports features that are not (currently) covered by OMG-IDL. These stem from the TINA computational architecture and include multiple interface objects, stream interfaces and Quality of Service (QoS) descriptions. A comparison of ODL and OMG-IDL can be found in Appendix C.

ODL has evolved over time. The Core Team developed information and computational specifications for TINA Services in 1994. An early version of ODL [6] was used for the computational specification of (the components of) these services [18] [19]. Analysis of how ODL was used in this endeavour, resulted in its revision, ODL v1.3. It was anticipated that ODL would continue to develop. The notion of Capability sets were introduced to distinguish significant changes to the language from incremental changes. For example, versions of ODL prior to, and including, v1.3 are known as Capability Set 1, or CS1. The expressive power of ODL has increased considerably since CS1. This document presents the most recent version of ODL, v2.3, which along with the other 2.x versions of ODL, is also known as Capability Set 2.

It should be noted that this document is intended to be a manual for ODL. It provides a syntactic description of the language using BNF (see Appendix A for a description of the notation). The architecture that provides the semantics of the language is mostly presented in additional TINA-C documents. As a starting point, "TINA Computational Modelling Concepts" [5] and "Object Grouping and Configuration Management" [23] are seen as foundational sources of this semantic information. Other relevant TINA-C architectural documents are introduced in the course of this work.

1.1 Document organization

Section 2 (Scope and Structure of ODL) presents the rationale for the creation and use of a language such as ODL.

Section 3 (Foundations of ODL) describes the overall structure of ODL and related it to the semantic basis of the syntax. Syntax is introduced in illustrative examples.

Section 4 (Syntax of ODL) describes the syntax of ODL, giving examples and providing brief explanations of the use of the language.

Section 5 (How to use ODL) describes the way ODL specifications are used for TINA application development.

Appendix A (BNF description of ODL) presents the complete syntactic rules for ODL using the BNF notation.

Appendix B (Further ODL development) presents material relevant to possible future versions of ODL. This material is under study but not mature enough to incorporate in this version of ODL.

Appendix C (Comparison: ODL and OMG-IDL) presents a comparison between ODL and OMG-IDL.

1.2 Main Inputs

Early versions of the TINA baseline document “Computational Modelling Concepts” [6][1] provided the initial definition of the language and the context for its mapping to engineering specifications. The 1994 TINA report “Mapping of TINA-C ODL to OMG-IDL” [3] provides useful suggestions for improvement of the language based on the feedback gained from the development of a translator from ODL to OMG-IDL. A report from the PLATyPus Auxiliary Project, “Programming tools for the PLATyPus experiment” [28], and an associated paper “PLATyTools and ODL” [29], also provided suggestions for improving ODL. Most of the suggestions from these documents have been incorporated in version 1.3 of this document.

A number of the concepts outlined in four Engineering Notes have also been incorporated into this document. The first Engineering Note, “What to do with ODL?” [10], provides an analysis of the usability of ODL and proposes some improvements. The second, “ODL Inheritance Rules” [11], proposes new inheritance and name scoping rules for ODL. A third engineering note, “ODL Syntax for Streams -- a description of the options” [12] introduces *stream interfaces* to specify continuous bit rate data (*flows*), and describes some possible syntaxes for flow types and the use of stream interface templates. The fourth Engineering Note, “Object Grouping and Configuration Management” [23] defines a syntax for object grouping which was adapted for inclusion into this document.

1. There are latter versions of this document [5] [27].

2. Scope and Structure of ODL

The first and second parts of this section present the motivation and scope of ODL. The third part introduces the structure of ODL syntax, and relates it to the underlying semantics.

2.1 TINA and ODL

The initial motivation for developing ODL was to enable the computational specification of ubiquitous TINA architectural concepts: *interfaces*, *objects*, and *object groups*. Specifications of these individual concepts are combined to form the (computational) specification of TINA systems. When enterprise, information, engineering and other specifications are added, the initial steps in the TINA software development process are underway.

This document primarily deals with the syntax of ODL. The architectural concepts and principles underlying ODL are documented separately. These architectures provide semantics for the ODL syntax. For example, the syntactic specification of (multi-interface) objects are documented here, while the semantics underlying the syntax is documented in “Computational Modelling Concepts” [5].

A number of existing syntaxes are capable of supporting aspects of TINA computational specifications. Examples include ANSA-IDL, DCE-IDL and OMG-IDL. Among them, OMG-IDL was chosen as the most appropriate base for extension within TINA-C. A significant factor in its adoption was that it complies with the object-oriented foundation of the TINA-C architecture. OMG-IDL also enjoys a rapid growth in industry acceptance. Consequently, ODL is based on OMG-IDL, with the intention that ODL be a superset of OMG-IDL.

The reader should note that OMG-IDL and other IDL languages do not fully support the concepts defined in the TINA computational model. For instance, the concepts of groups, stream interfaces, and service attributes are not supported by existing IDLs. ODL extends OMG-IDL, providing support for these concepts.

The reader should also note that ODL is primarily a means of specifying types. Consequently, this document mainly deals with types. Occasionally, instances are referred to. The reader should be aware of the distinction, and take care to interpret descriptions as referring to types or instances as context dictates.

2.2 Objectives

The design goals for ODL center upon three objectives. These are to provide a language for:

1. Application specification and specification re-use (at development time).

   When developing computational specifications, an application developer needs to be able to describe a TINA application in terms of computational modelling concepts, such as object groups, computational objects, operational and stream interfaces, and data types. In addition, development effort can be reduced if existing computational specifications of TINA applications can be reused in the specification of new applications.

2. CASE tool development.

   Application specification, and the application development process, can benefit from the kind of automation typically offered by CASE tools. In order to begin constructing such CASE tools, the language of specification needs to be available and supportive.

3. The type specification of application components, as needed at run time.

   In order to support dynamic binding to interfaces, and dynamic configuration management of systems, a language is useful for describing the entities involved, and for performing type checking functions. The availability of a language to describe operational and stream interface types aids in the definition of compatibility for interfaces. For example, when binding to a remote interface, knowledge of its type is useful. Supervisory and control systems for distributed applications are also aided by the availability of type (and implementation) repositories.

ODL is required to support these three categories of requirements.

The following are not objectives of ODL development, and are characterized as non-goals:

1. To provide a language for describing the complete specification of a TINA system.

   It is not the objective of ODL to provide support for the complete specification of computational entities. For example, constraints specified from the TINA Enterprise and Information Viewpoints are not intended to be completely represented in ODL. An ODL specification may be augmented using additional formalisms. For example, no formalisms are presented for the detailed behavior specification of a computational object. Formalisms like *message sequence diagrams* (also known as *object interaction diagrams*), *SDL-92*, or *LOTOS*, can be used to augment ODL for this purpose.

2. To define a programming language for object implementation.

   ODL is primarily a specification language. When developing programs from ODL, the facilities of existing programming languages, such as C++ and Smalltalk, are required. This is clearly seen when we consider the abstraction facilities supported by ODL (see Section 3). Data abstraction is supported by the rules that support data types. This allows simple data types to be combined into complex data types. However, procedural abstraction is not supported in ODL[1]. Simple procedures cannot be combined into complex procedures. Operational interfaces only allow a procedure's signature to be stated, not its composition. The procedural abstraction mechanism of a programming language needs to be added to ODL when developing applications.

3. To prescribe a language for describing TINA Engineering Modelling Concepts.

   ODL is designed to describe computational specifications, and it is not primarily intended to support engineering modelling concepts.[2] However, it may be used for this purpose if a mapping between Computational and Engineering viewpoints is

1. It should be noted that syntax for flow abstraction is not present in ODL either.

2. Although it must be noted that some concepts supported by ODL (such as the "object execution model" which indicates that objects interact by sending and receiving messages or are connected by continuous bitstream pipes) have closely related computational and engineering modelling aspects.

documented. Such a mapping is undertaken, either implicitly or explicitly, within any system development process.

2.3 Overall structure of TINA ODL

ODL (syntax) is composed of five major parts: *data types and constants*, *stream interface templates*, *operational interface templates*, *object templates*, and *object group templates*. These, and their inter-relations are shown in Figure 2-1. The most fundamental part of ODL supports data type and constant definitions. These definitions can be made at any point in ODL specifications as long as names are defined before use. Operational interface templates are used to specify procedures in ODL (as interface templates do in OMG IDL). Stream interface templates are introduced to specify continuous-bit-rate data (flows). Object templates represent the basic unit of distribution in ODL; namely object instances. They incorporate stream and operational interface templates. Object group templates enable aggregation of object templates to increase the conceptual level at which programs can be designed and increase the modularity of designs.

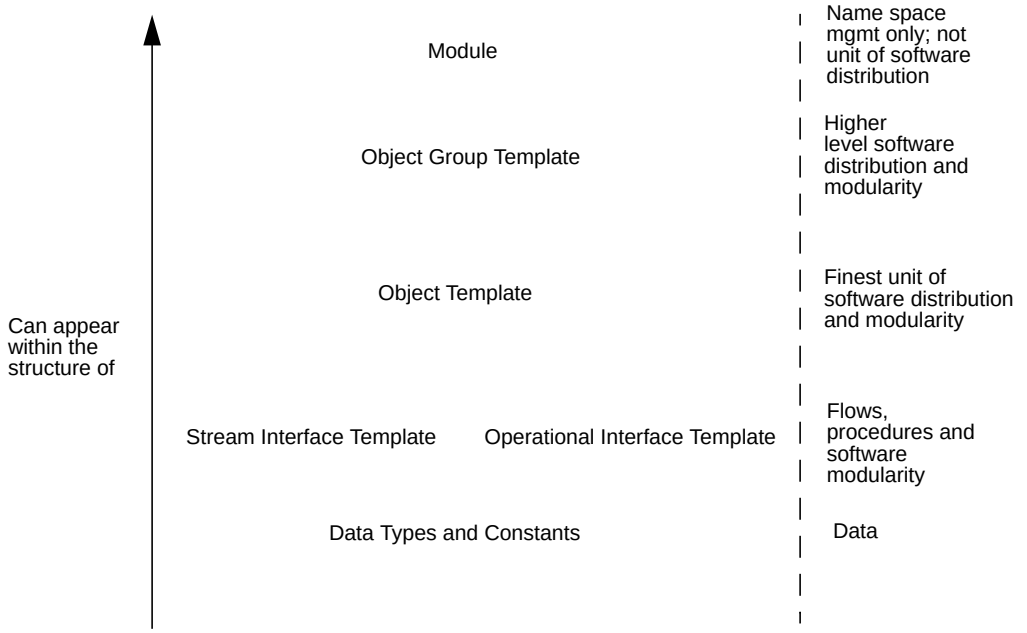


Figure 2-1. Overview of the semantic basis of ODL syntax. Note that ODL is intended to be used by CASE tools, and all ODL types are intended to be units of compilation.

The semantics that underlies ODL syntax is documented in a number of places. Understanding these relationships is the key to understanding the structure of ODL and its future evolution. Data type and constant syntax is directly based upon OMG-IDL [37], hence takes on its semantic associations.

The semantic basis of the four templates (stream interface, operational interface, object and object group) is more diverse and can be seen to have 2 forms of origin. The first form of semantic basis is related to the intrinsic (compositional / signature / core) aspect of the templates. For example, operational interface templates basically contain syntax for aggregating procedures, stream interface templates basically contain syntax to aggregate *flows*, object templates can be seen to contain syntax composing operational /stream interfaces and state, while object group templates basically contain syntax to aggregate object templates. For stream and operational interfaces, and objects, the semantics of this composition, or aggregation, stems from the architecture depicted in “Computational Modelling Concepts” [5]. For object groups, the semantics of this composition, or aggregation, stems from the architecture originally depicted in “Computational Modelling Concepts” [5] and as extended in “Object Grouping and Configuration Management” [23]. This first level of relationship between syntax and architecture for templates is depicted in Figure 2-2.

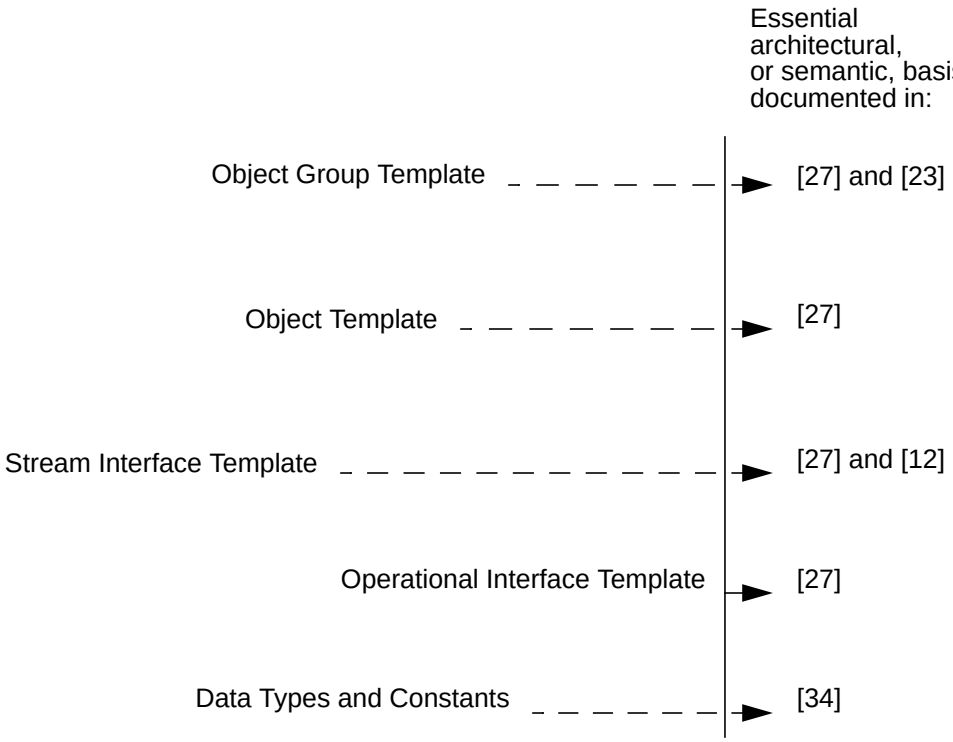


Figure 2-2. Diagrammatic overview of the semantic basis for the core ODL syntax.

The second form of relationship between template syntax and semantics assumes the first is in place and builds upon it. The relationships can be understood in terms of the following two step ideal.

1. An architecture for some aspect of TINA software is devised and documented. For example an architecture for reuse through inheritance.

2. The concepts and principles of this architecture are then reflected in additional ODL language terms, as needed. For example, ODL terms that support inheritance in interfaces, objects and object groups.

Unfortunately the evolution of ODL has been more organic than this “ideal”, resulting in the architectural basis of the syntax being documented in a number of places, as depicted in Table 2-1[3].

Table 2-1. Showing the documentation of ODL template sematics. The rows indicate the architecture expressed as ODL syntax, while the columns indicate the major ODL structures. The table entries indicate the documents containing the architecture underlying the syntax of the corresponding ODL structure. Table entries indicate either: documents listed in the references section of this document, [*] refers to this document, N/A means not applicable, U/D means future development.

Architecture	Architectural / Semantic basis of ODL syntax			
	Stream Interface Template	Operational Interface Template	Object Template	Object Group Template
Initialization and Configuration	[27]	[27]	[27]	[23]
Behavior	[*]	[*]	[*]	[*]
Quality of Service	[*], [26]	[*], [26]	N/A	N/A
Reuse through Inheritance	[*], [11]	[*], [11]	[*], [11]	[*]
Trading	[*], [24]	[*], [24]	N/A	N/A
Transaction	U/D	U/D	U/D	U/D
Security	U/D	U/D	U/D	U/D

On Table 2-1, the row titles indicate architectures identified through a retrospective look at previous versions of ODL. The structure of the current version of ODL directly reflects these architectural bases. We will briefly present the scope of these architectures below:

- The Initialization and Configuration architecture provides concepts and principles related to the creation and startup of new ODL template instances. For example, issues of whether there should be a “special” management interface on each object (for configuration purposes), and if so, which entities are expected to use it, and whether it has an initialization operation on it, and what the scope of it is, and so forth, are part of such an architecture.

3. If future additions to ODL syntax are made, they should follow the “ideal” development process to a greater extent, thereby more simply documenting the relationship between ODL syntax and semantics.

- Behavior of an entity can be thought of as the complete set of possible interactions with the entity[4]. The architecture for behavior deals with how those interactions are documented, the extent of the documentation (which may also depend upon context), the types of interactions of interest, and so forth.
- The Quality of Service architecture deals with the nature of guarantees that an entity provides to other entities communicating with it.
- The Reuse through Inheritance architecture is concerned with how to re-use specifications through a subtyping paradigm. It indicates what is re-used and makes explicit the rules of inheritance.
- The Trading Architecture deals with how remote entities locate each other in a distributed environment. Elements of this location process can be supported by ODL.
- The Transaction Architecture deals with support of ACID[5] (or more relaxed) constrained communication. The various features of the architecture (for example whether less than transactional semantics are defined) reflect upon how the syntax captures those features.
- The Security Architecture deals with authorization, authentication and encryption/privacy issues associated with TINA based systems.

By viewing ODL according to the above structure, the possibilities and means for its extension are explicit. Either the existing architectural bases are extended, and corresponding syntax added, or additional architectural bases documented, and corresponding syntax added[6]. As an example of the latter case, if security constructs need to be added to ODL, then a a security architecture should be specified, and ODL modified to support that architecture.

4. The behaviour described in ODL templates should be the minimal description required to use the interface (in the case of interface behaviour) or to manage the object (in the case of object behaviour). An implicit assumption in all behavioral specifications would be that implementation issues which do not impact on the use of these templates would not be raised in behaviour specifications. This is particularly important in terms of inheritance, where specialization should not extend the behavior of an object or interface.
5. ACID is an acronym for the following transactional properties: atomicity, consistency, isolation, durability.
6. The authors recognize that this is a simplified view, as there is the possibility that syntax is changed with no architectural modification and that an element of syntax may be dependant upon multiple architectures.

3. Foundations of ODL

This section overviews the foundations of ODL; its meta structure and semantics. During the course of this section ODL syntax is presented for the purposes of illustration. The meaning of this syntax is explained as it is introduced, but the reader is reminded that a detailed presentation of ODL syntax is the subject of the following section.

Initially in this section, basic definitions and conventions are presented. Following this, the rules for basic naming and scoping are introduced. Further naming and scoping rules are added along with each architectural addition. A feature of ODL is its ability to share and re-use existing specifications. One way in which re-use is accomplished is by building specifications through composition. The principles upon which this is done are presented in Section 3.3. When a new ODL entity is instantiated it may need to be initialized, and make itself known in the distributed environment. This initialization and configuration architecture is presented in Section 3.4. Following this we present the principles upon which behavior and quality of service are specified. As stated above, one form of specification re-use is through composition, while a second form is through specialization or inheritance. The specific inheritance architecture adopted in ODL is presented in the subsection 7. The final section deals with the trading architecture as it pertains to ODL.

Throughout this section main points are highlighted by labels of the form Rn, (eg. R1, R2, etc). Definitions of terms are indicated with labels of the form Dn (eg. D1, D2, etc).

3.1 Definitions and Conventions

3.1.1 Definitions

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

D1 Component: An ODL *component* is either an object group template, object template, interface template, operation, or flow. The signature specification of an ODL component is declared in ODL.

D2 Supporting definition: Definitions of data types, constants, and exception declarations, are called *supporting definitions*.

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

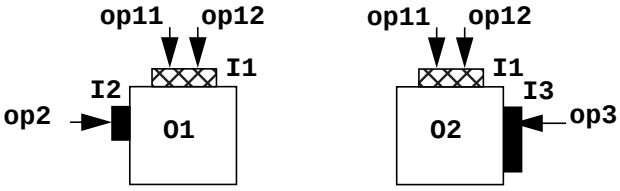
3.1.2 Graphical Conventions

The following conventions are applied to the graphical representation of the examples given in the remainder of this document. *It should be noted that the diagrams are of types, and should not be confused with diagrams of instances which appear in other TINA-C documentation*:

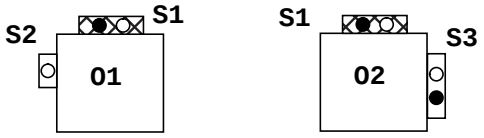
- Objects are represented as boxes (rectangles).

- A supported operational interface is represented as a filled rectangle contiguous with the box representing the object to which it belongs. Some interfaces might be emphasized by the use of a special pattern.

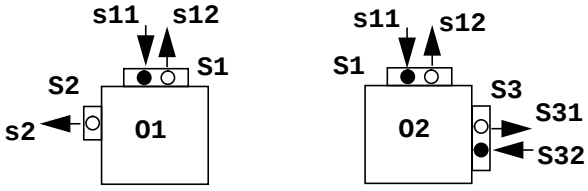
- An operation is represented as an arrow pointing to the operational interface to which it belongs. Other elements of operational interfaces are not represented.



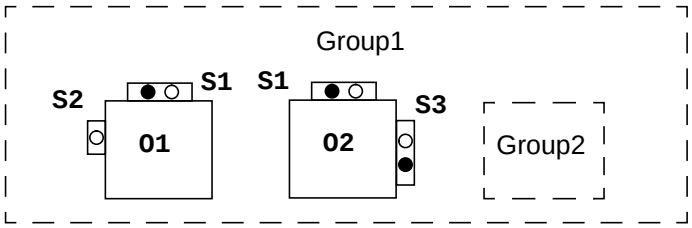
- A supported stream interface is represented as a rectangle containing open and/or filled circles, contiguous with the box representing the object to which it belongs. Some interfaces might be emphasized by the use of a special pattern.



- A flow sink is represented as an arrow pointing to a filled circle on the stream interface to which it belongs. A flow source is represented as an arrow pointing away from an open circle on the stream interface to which it belongs. Other elements of stream interfaces are not represented.



- Object groups are represented by dashed line boxes. Containment in an object group is represented by containment in a dashed line box.



Having introduced the basic terms and graphical conventions used throughout this document, we will now examine the naming and scoping framework of ODL.

3.2 Naming and Scoping

Naming and scoping rules are defined to enable the unambiguous identification of components. For a comparative analysis of related work, see also comparable rules in OMG-IDL ([37] p. 3-31).

R1 An entire ODL file forms a naming scope.

R2 The following kinds of definitions form nested scopes within a file[1]:

- Module
- Object Group
- Object
- Interface
- Structure
- Union
- Operation
- Exception

For example, the following ODL definitions are contained in one file. It specifies a module, `M1`, containing an object group, `G1`, containing an object, `O1`, containing an interface, `I1`, containing a data type, `dataType1`, and an operation, `operation1`[2]. `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`[3].

1. A module cannot span files.
2. An interface can either contain operations (operational interface) or flows (stream interface), but not both.
3. Note that in practice it would be typical to define groups, objects and interfaces within the scope of a module or within the global scope, rather than in the style of strict nesting shown here. Defining entity types within the scope of other types, as shown here, hampers reuse of specifications, as discussed in Section 3.3.

```
module M1 {
   ...
   group G1 {
      ...
      object 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 Groups
- Objects
- Interfaces
- Operations
- Data Types
- Constants
- Enumeration values
- Exceptions
- 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 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 ODL below shows dataType1 defined in the scope of module M1. It is then used (as an unqualified name) within the scope of interface 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 ODL definition in a file has a *global name* within that file.

The rule to create a global name is the same as in OMG-IDL ([37] p. 3-32):

"Prior to starting to scan a file containing an OMG 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 OMG 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* and *object* are treated similarly to *interface*, *struct*, *union*, and *exception***.**

Having introduced some of the basic components of the ODL language, we will now look at how ODL specifications can be presented as documents, particularly with a view to reusing ODL specifications.

3.3 Interface, Object and Object Group Separation and Sharing

3.3.1 Motivation and Overview of Separation

Freedom is offered to the developer of computational specifications for independent declaration of interfaces, objects and object groups[4]. Each interface template in ODL may be self-contained, and may be reused in any number of object templates. Similarly, object templates may be specified as individual units, and reused in any number of object group templates. This is shown diagrammatically in Figure 3-1.

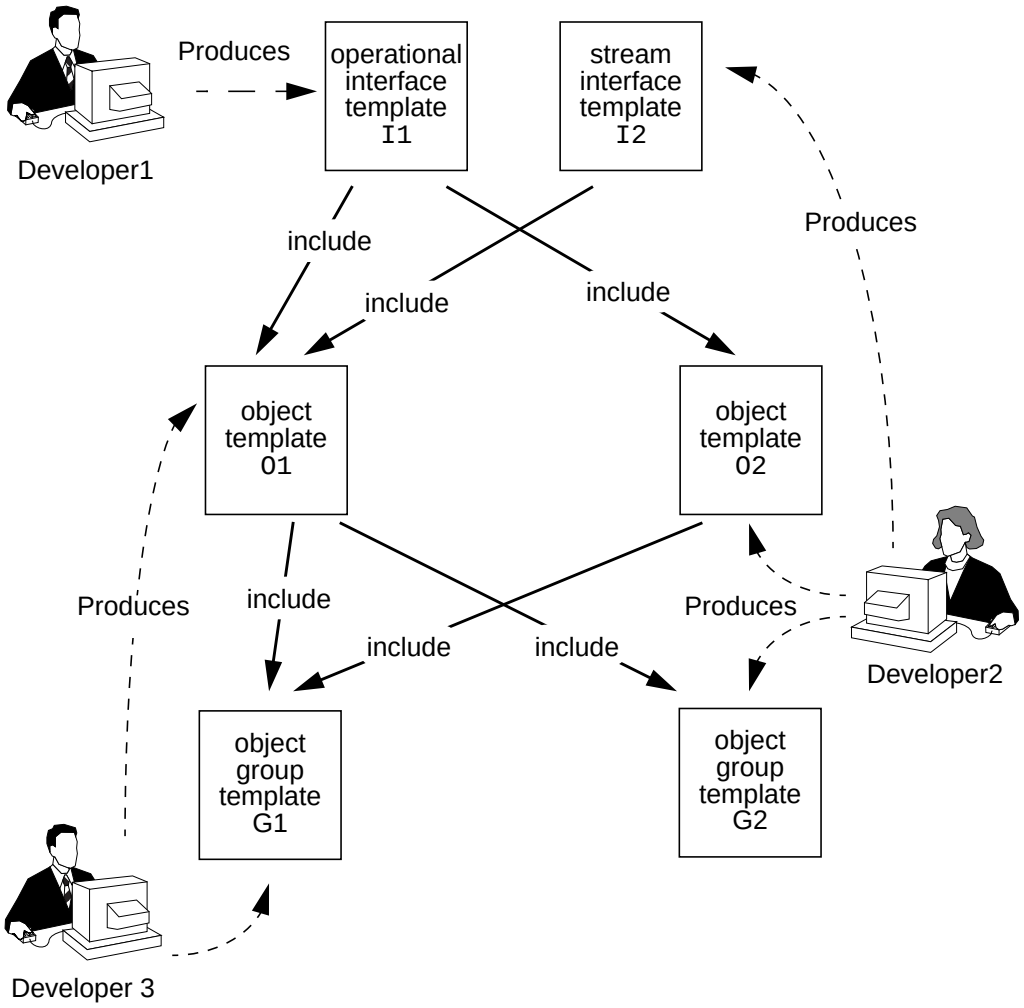


Figure 3-1. Showing flexibility in the development of ODL specifications.

4. The independence of interface templates and object templates in ODL v1.3 represents a major difference from earlier versions of ODL, where the operations supported by an object were defined directly in the object template, and the interface declarations served only to group operations.

In Figure 3-1 for example, the two objects, O1 and O2, share a common definition of interface I1. Using a single specification of I1 helps ensure a consistent definition of I1. The interface I1 itself may have been developed by an unrelated developer, but may define a service which objects O1 and O2 are to provide. I1 is being reused by appearing in both object templates. The consistency and reuse of object templates with respect to object group templates follows similarly.

As I1 is an operational interface and ODL is a superset of OMG-IDL, the mapping between the I1 template and an equivalent OMG-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 OMG-IDL interface definitions.

3.3.2 Sharing of Component Declarations

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

3.3.2.1 Data type Declaration Sharing

Data type declaration sharing is intended for the sharing of ODL specified data types between several components.

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

3.3.2.2 Operation Declaration Sharing

Following OMG-IDL syntax, the following rules apply to 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 interfaces is not possible.[5]

Remark: This rule is directly derived from OMG-IDL. A new version of OMG-IDL may relax this rule. The implication of such a change for ODL is an open issue.

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

5. Note that it is possible to re-use an operation signature declaration by inheriting from an interface template declaring it.

3.3.2.3 Flow Declaration Sharing

The following rules apply to 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 interfaces is not possible.[6]

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

3.3.2.4 Interface Declaration Sharing

It is assumed that interface declaration sharing is intended for the sharing of an ODL specification of an interface between several objects. ODL syntax is defined to enable the separation of interface declarations from object declarations. Interface specifications can be included in an object template declaration as *supported interfaces* or as *required interfaces*. Supported interfaces can exist on instances of their encompassing objects.

D3 Declared supported interfaces / offered interfaces: Interface types listed as being supported on an object's template (declared *supported interfaces*) are the only interface types for which instances may be accessed on instances of the object[7]. The *offered* interfaces of an object instance are the interfaces existing on that object instance.

D4 Declared required interfaces: The declared *required interfaces* on an object template lists some of the types of interfaces which an instance of the object template can invoke operations upon[8].

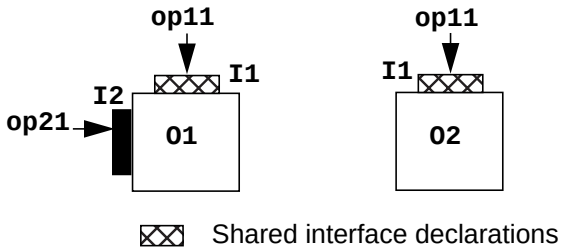
The rule are as follows:

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

As a simple example, assume there are two object templates, `O1` and `O2`, and that they contain the one common interface template, `I1`, and the separate interface template `I2`. This situation is depicted in the diagram below.

6. Note that it is possible to re-use a flow signature declaration by inheriting from an interface template declaring it.
7. The declared supported interfaces on a *base class* (see Section 3.7) are considered (not declared) *supported interfaces* on the inheriting class (class doing the inheriting).
8. The (declared) required interfaces on a *base class* (see Section 3.7) are considered (not declared) *required interfaces* on the inheriting class (class doing the inheriting).

As a simple example, assume there are two object templates, O1 and O2, which contain the one common interface template, I1, and that O1 also contains the separate interface template I2. This situation is depicted in the diagram below.



It is possible to declare the interfaces in the manner shown below[9]. Note that declarations made between a pair of horizontal bars are considered to be in one file[10]. The general syntax for an operational interface template is “interface <interface_name> { <interface body> }”. The interface body generally consists of a preamble followed by a list of operations.

```
interface I1{
    ...
    typedef … DataType11;
    typedef … DataType12;

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

interface I2{
    ...
    typedef … DataType2;

    void operation21 (in DataType2 …);
    ...
}; // end of I2
```

It is possible to declare objects O1 and O2 in the manner shown below. The general syntax of an object template is “object <object_identifier> { <object body> }”. Inside the object body, interface identifiers listed after the keyword “supports” indicate the declared supported interfaces of the object.

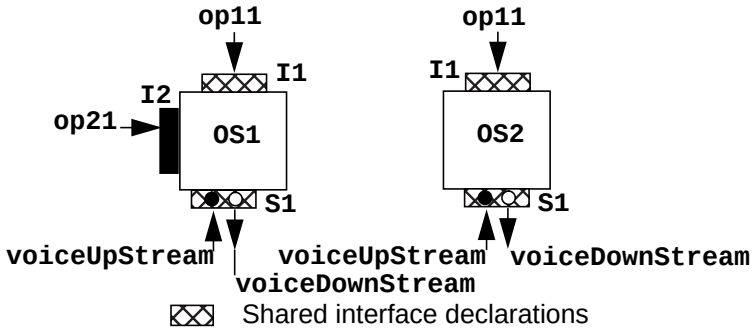
9. To increase readability, operations indicated as “op...” on diagrams are indicated as “operation...” when written in ODL code segment. For example op11 indicated in the diagram above is written as operation11 when referred to in the corresponding code segment below.

10. Data type, interface, object and group identifiers are viewed as having global scope if there is no scoping construct between them and the “file”.

```
object O1{
   ...
   supports
      I1, I2;
   ...
}; // end of O1

object O2{
   ...
   supports
      I1;
   ...
}; // end of O2
```

We will now extend the previous example to include sharing of stream interfaces. Assume each of the above object templates, O1 and O2, now also share a common stream interface template S1, as depicted in the following diagram.



Assume also that each operation and flow is optionally associated with a quality of service specification (for more detail see Section 3.6). The operational interfaces, `I1` and `I2`, will be similar to those declared above, while `S1` is newly defined below. Each operation and flow is followed by the keyword “with”, which in turn is followed by a datatype, and variable identifier, specifying parameters describing the quality of service associated with the operation or flow[11]:

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

11. Note that values of QoS types are not specifiable in the current version of ODL. This is considered a serious limitation, expected to be remedied in future versions of ODL. When using the current syntax, the QoS value associated with the QoS type is expected to be assigned, or negotiated, when the interface is instantiated. It should also be noted that the semantics of each QoS type is expected to be documented as text comments in this version of ODL.

```
    // QoS type
    // max time server allowed for operation completion
    typedef … ServerResponseTime;

    void operation11 (in DataType11 …, out DataType12 …)
       with ServerResponseTime opResponseTime11;
    ...
}; // end of I1


interface I2{
    ...
    // data types
    typedef … DataType21;

    // QoS type
    // max time client will wait for operation completion
    typedef … ResponseTime;

    void operation21 (in DataType21 …)
       with ResponseTime opResponseTime21;
    ...
}; // end of I2
```

The stream interface S1 may be declared as follows (for the purposes of illustrating stream declarations, we will assume that S1 supports a voicemail service):

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

    // QoS type
    typedef … VoiceQosType;

    source VoiceFlowType voiceDownStream
       with VoiceQosType voiceDownStreamQos;
    sink VoiceFlowType voiceUpStream
       with VoiceQosType voiceUpStreamQos;

}; // end of S1
```

It is possible to declare the new objects OS1 and OS2 in the following manner:

```
object OS1{
    ...
    supports
       I1, I2, S1;
    ...
}; // end of OS1
```

```
object OS2{
   ...
   supports
      I1, S1;
   ...
}; // end of OS2
```

3.3.2.5 Object Declaration Sharing

It is assumed that object declaration sharing is intended for the sharing of an ODL specification of an object between several groups. ODL syntax is defined to enable the separation of object declarations from group declarations. Object specifications can be included in a group template as *components*. Interface specifications can be included in an object group template as *contracts*, which are interface types that can be used by entities external to the object group.

D5 Declared component objects: The declared *component* entities of an object group template are the object types, or object group types, which can be directly instantiated by the manager object instance of an object group instance. The declared component entities are listed as “components” on a group template.

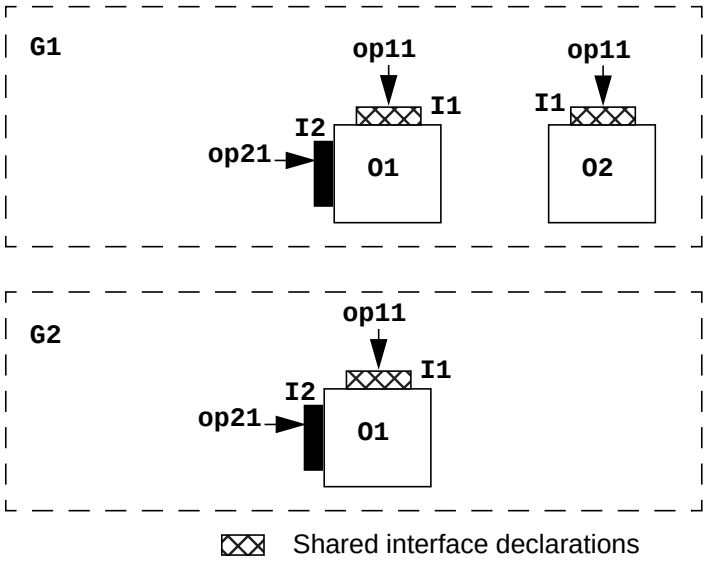
An object group instance is integrally associated with the manager object instance (group manager). It is expected that group managers are in the same cluster as their component object instances, while component group instances may be in a different cluster (and hence potentially on a different node). Upon object group instantiation, the initial configuration (hard coded into the group manager, passed to the group manager, read from a file or repository by the group manager, or obtained by other means) dictates which components are initially instantiated.

D6 Declared contract: A declared contract of an object group template is one of the interface templates of a component object of that object group template. The declared contracts on an object group template represent the only interfaces able to be used by entities external to an instance of that object group template. The declared contracts are listed as “contracts” on a group template.

The rule is as follows:

R16 Object templates can be declared outside object group templates.

As a simple example, assume there are two group templates, G1 and G2 that each contain the one common object template, O1, and separate object template O2. The objects O1 and O2 are as defined initially in the previous section. This situation is depicted in the diagram below.



It is possible to declare the groups in the manner shown below (assuming the objects `O1` and `O2`, and interfaces `I1` and `I2`, are as defined initially in the previous section). The general syntax for an object group template is “group <group_name> { <group body> }”. Inside the group body, identifiers listed after the keyword “components” indicate the component objects (and if applicable object groups) of the object group. Interface identifiers listed after the keyword “contracts” indicate the interfaces of the object group that are visible to entities outside the group.

```
group G1{
    ...
    components
        O1, O2;
    ...
    contracts
        I2;

}; // end of G1

group G2{
    ...
    components
        O1;
    ...
    contracts
        I2;

}; // end of G2
```

3.3.3 Naming and Scoping with respect to Sharing/Separation

The following scoping rules are relevant to shared specifications.

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

In the following example, there are two interfaces, `I1` and `I2`, and two objects `O1` and `O2`. `I1` is defined in its own scope in a separate file, while `I2` is defined in the scope of `O1`[12]. `O1` and `O2` are defined in separate files. Both object `O1` and `O2` use interfaces `I1` and `I2`. Note that it is possible to declare interface `I2` within `O1`, before declaring that `O1` will support it. Note also that in the declaration of object `O2`, the identifier of the interface `I2` is qualified by the scoped name `O1::`.

```
interface I1{
   ...
}; // end of I1
```

```
object O1{
   ...
   // inline definition of I2
   interface I2{
      ...
   }; // end of I2
   ...
   supports
      I1, I2; // supported use of I1
   ...
}; // end of O1
```

```
object O2{
   ...
   supports
      I1, O1::I2;
   ...
}; // end of O2
```

12. While it is permissible in ODL to define interfaces in the scope of objects (and objects in the scope of groups), this style should be used with caution, as reuse of specifications is made somewhat complicated.

R18 In the scope of an object group, an object identifier can be *defined* by declaring the associated object template inline, or *used* by declaring it as a component. In each case, the global name of the object template will be different.

R19 In the scope of an object group, an interface can be *defined* by declaring the associated interface template inline, or *used* by declaring it as a contract. In each case, the global name of the interface will be different.

The following example extends from the previous one. There are two interfaces, `I1` and `I2`, two objects `O1` and `O2`, and two groups `G1` and `G2`. `I1` is defined in its own scope in a separate file, while `I2` is defined in the scope of `O1`. `O2` is defined in a separate file. `O1` is defined in the scope of `G1`. `G1` and `G2` are defined in separate files. Both object `O1` and `O2` use interfaces `I1` and `I2`. Both group `G1` and `G2` contain objects `O1` and `O2`. Note should be made of the naming complexity that is demonstrated in this example. Greater readability and flexibility would be achieved if each interface object and group template had top level scope and were defined in their own files.

```
interface I1{
   ...
}; // end of I1
```

```
object O2{
   ...
   supports
      I1, G1::O1::I2;
   ...
}; // end of O2
```

```
group G1 {
   object O1 {
      ...
      interface I2 {
         ...
      }; // end of I2
      ...
      supports
         I1, I2;
      ...
   }; // end of O1
   ...
   components
      O1, O2;
   ...
   contracts
      I1, O1::I2;

}; // end of G1
```

```
group G2 {
    ...
    components
        G1::O1, O2;
    ...
    contracts
        I1, G1::O1::I2;

}; // end of G2
```

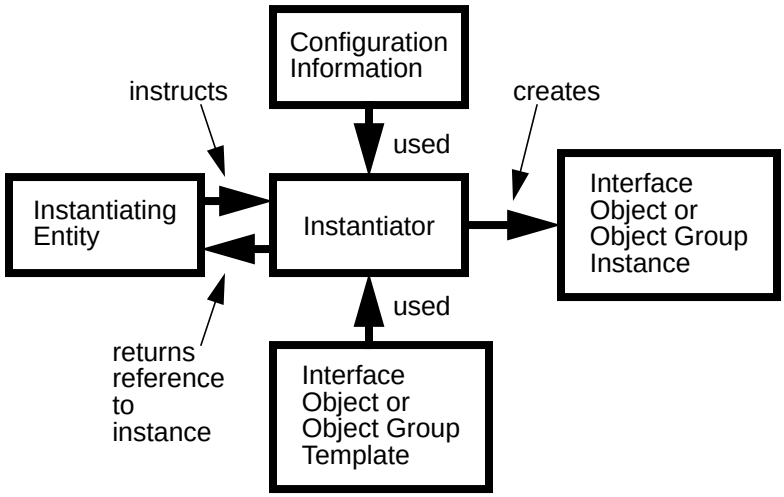
3.4 Initialization and Configuration

Interface, object and object group instantiation is generally associated with a sequence of activities. In this section we present the concepts and principles underlying these sequences for ODL.

In general when an interface, object or object group template is instantiated, an

- instantiating entity,
- agent of instantiation (instantiator), and
- initial configuration specification[13]

are required. This is shown in a representative diagram below.[14] One example of an instantiating entity (the entity that requests the instantiation) is an object. The instantiator may be the programming language implementation itself (the creation process being a language type instantiation), and the configuration information may be implied in the code implementing the template.



13. The means of specification and run-time location of initial configuration information is for further study.
14. It should be noted that this is an abstract depiction.

With respect to creating and initializing an entity, the following applies:

R20 An object group template must specify one object template as being the *manager* object.

When an object group template is instantiated, a *manager* object is expected to be automatically created, and an (operational) interface reference to its initial interface (see R21) returned to the instantiating entity. Typically this interface and manager object would be used to initialize and manage the object group instance.

Following is an example of the use of the manager syntax.

```
group OG1 {
   ...
   components
      O1,O2,O3;
   manager
      O1;
   contracts
      I1,I2;
   };
```

R21 An object template must specify one (operational) interface template as being the *initial* interface.

When an object template is instantiated, an initial operational interface is expected to be automatically instantiated, and an interface reference to it returned to the instantiating entity. Typically this interface would be used to initialize and manage the object instance[15] [16]. For example, an object factory that instantiates an object template will recieve an interface reference to the initial interface, which it should then pass onto the entitiy that requested creation of the new object. As another example, when a group manager instantiates a component object, it recieves an interface reference to its initial interface.

Following is an example of the use of the initial interface syntax.

```
object O1{
   ...
   supports
      I1, I2;
```

15. It is possible that the thread of control to initialize an object is provided via an operation on the initial interface (eg. an init operation), or provided by the programming language's creation mechanism (eg. constructor methods on C++ objects).

16. It is expected that the initial interface allows navigation to the other interfaces of the object.

```
    initial
       I1;
}; // end of O1
```

3.5 Behavior

The behavior of an entity, in its most general sense, consists of all the possible interactions the entity can undertake with its environment. ODL is not sufficiently mature to provide a complete and detailed specification for behavior in this sense[17]. Instead, an informal behavior specification is described for particular entities as follows:

- Interfaces:

  This specification describes the service provided by an interface of the type being defined. It also specifies an informal *usage* specification. This specification documents the ordering (or sequencing) constraints on the operations defined in the interface template. Invocations of operations on an instance interface of such a template must satisfy these constraints. In the current version of ODL this specification is a string literal, although more formal description techniques may be supported in future revisions of ODL.

- Objects:

  This specification describes the responsibilities of an object in providing services via each of it's interfaces as supported in ODL[18]. It also specifies (declared) required interfaces types which will be used by the object to perform its functions and provide its services.

- Object groups:

  This specification describes the responsibilities of an object group in providing services via each of it's contracts[19].

Currently in ODL, behavior specifications have structure. There is a component that supports specifications documented in any form delimited as a text string, and natural language (e.g. English) is suggested. Additionally, interface behavior specifications support a similar text string for documenting usage specifications, and object behavior specifications support "requires" keyword which is followed by a list of required interface types.

An example of a behavior specification for an interface is shown below[20]:

17. The definition of a detailed syntax for behavior specification is left for further study. A candidate for such a syntax has been defined in "Quality of Service Framework" [14].

18. When writing this, and the interface specifications, the developer should be aware that the object template will almost certainly not be available to users of the services provided by the object. It is important that all notable information on services be described in the interface templates, not the object template.

19. When writing this, and the interface specifications, the developer should be aware that the object group template will almost certainly not be available to users of the services provided by the object. It is important that all important information on services be described in the interface templates, not the object or group templates.

```
interface CSMConfiguration: Management::ServiceManagement {

   behavior
          behaviorText
          “This interface serves to configure the object 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.

          Concurrent calls to ReadState operations are permitted
          but all operation invocations are blocked while a
          WriteState operation is being handled.

          WriteState operations will block until all current
          ReadState or WriteState invocations are completed.”;
   ...
};
```

3.6 Quality of Service

Specification of the basic capabilities 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 quality of service associated with an operation or flow is the, timing constraints, degree of parallelism, availability guarantees and so forth, to be provided by that entity or component. Quality of service information deals with the provision of the service, rather than the service itself. Quality of service specifications allow state-

20. Note that not all compilers allow line breaks in strings. For these compilers an alternative means of declaring a string longer than a line is to specify sequences of strings such as “...” “...”, which are treated as one large string by the compiler.

ments about the “level of service” offered by components. In general this information may be provided at component specification time, or dynamically by the management system responsible for initiating entity creation. It may even be altered during the lifetime of the service offering.

The problem of adding quality of service specifications to 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 = 3Mbits` may mean that the interface with that flow cannot be offered unless it can source/sink exactly 3Mbits, or typically connections to the flow are made at 3Mbits, but other values may be negotiated, or the maximum bandwidth of a connection is at 3Mbits, and only values lower may be negotiated, and so forth. The particular quality of service sematics to be supported in ODL has not been finalized.

In this version of ODL, semantics is left to the programmer. ODL allows for a quality of service type and variable identifier with any of[21]:

- operation
- flow

Values are not ascribed to the quality of service variables at specification time. Instead they are to be assigned at instantiation time[22]. The semantics are programer dependant. This capability is acknowledged as being severely restricted.

Following is an example of an operational quality of service specificaton. Note that the quality of service parameters are added after the keyword “with”. `operation1` is expected to be offered with a quality of service that is described using an instance of `BoundedResponseTime`. Note that the identifier of the variable instantiating `BoundedResponseTime, op1Qos` must be unique within the interface:

```
interface I1{
    ...
    typedef float BoundedResponseTime;
    ...
    void operation1 (in dataType1 var1)
       with BoundedResponseTime op1Qos;
    ...
}; // end of I1
```

Following is an example of a flow quality of service specification. Again 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.

21. Quality of service statements associated with interfaces, objects and groups are for further study.
22. Note that if there is a need to assign values, for the sake of recording this information, this can be done as comment statements in the present version of ODL.

```
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
```

Chapter 4 of the document “Quality of Service Framework” [26] presents an architectural framework useful for providing sematics to operation quality of service terms like r*esponse time*, *minimum invocation interval* and *maximum invocation interval*. These terms can be defined in terms of events associated with operation invocation, such as *invocation emission*, *invocation receipt*, *response emission* and *response receipt*. The reader is referred to this work as an input to defining quality of service parameters for operations. ODL does not specify standard quality of service parameters for operations.

3.7 Inheritance

3.7.1 Introduction and Motivation

Interfaces, Objects and Object Group templates are considered units of specification modularity. Since these 3 templates also represent types it is convenient to define a reusability mechanism where ([34] p.62):

1. A module can directly rely on the entities defined in another module (of the same kind of template). For example, a data type defined in one interface is used within another interface.
2. A type is derived from another type (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 ODL, defining rules for inheritance will allow new interface, object and object group templates to be declared as extensions or restrictions of previously defined ones.

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

3.7.2 Definitions

The following definitions are relevant to inheritance.

D7 Base / derived / specialized component: A component (group, object or interface) is said to be *derived from* or *specializing* another component, called a *base component* of the derived component, if it inherits from this base component.

D8 Most specialized: The *most specialized* objects (groups, or interfaces) within a set of objects (groups, or interfaces) are the elements of the set from which no other components within the set are derived (i.e., which are not the base of any other component of the set).

D9 Direct / indirect base: A component is called a *direct base* of a component (group, object or interface) if it is mentioned in the inheritance specification of the component declaration, and an *indirect base* if it is not a direct base but the base of a direct or indirect base (sub-inheritance).

D10 Inheritance graph / partial inheritance graph: The *inheritance graph* of objects (groups, or interfaces) is the directed acyclic graph representing the inheritance relationships between objects (groups, or interfaces). A *partial inheritance graph* of a given type (groups, objects or interfaces) is an inheritance graph restricted to a set of components.

Remark: The leaves of the inheritance graph for a given type of component (i.e., for groups, objects or interfaces) are the most specialized components of this type.

D11 Restriction / restricted set: The *restriction of a set* of components (set of groups, set of objects, or set of interfaces) is constructed by removing from this set any component which is the base of any other component from the set. The result of the restriction of a set is called a *restricted set*.

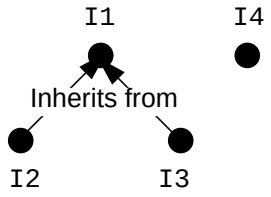
Remark: A restricted set can also be seen as the set of leaves of the partial inheritance graph.

Remark: The restricted set of all the components of type object (group or interface) is the set of most specialized objects (groups or interfaces).

Example A: Assume that the following interfaces with the following inheritance relationship are defined:

- Interface `I1`,
- interface `I2`, which inherits from interface `I1`,
- interface `I3`, which inherits from interface `I1`,
- Interface `I4`.

The following inheritance graph exists for interfaces:



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

3.7.3 Inheritance in Component Declarations

3.7.3.1 Interface Inheritance

It is assumed that interface inheritance provides “re-use by specialization” of (an ODL specification of) an interface. The interface that is inherited from is called a *base interface*. The interface that does the inheriting is called the *derived interface*. This specialization can take two forms:

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

  Remark: Note that the current version of OMG-IDL does not allow this kind of inheritance for operational interface templates (as reflected in rule 27), hence neither does ODL.

  Remark: Note that all derived interfaces of a base interface are considered “compatible with” the base interface.

The syntax defined for ODL fully supports OMG-IDL interface inheritance rules, and adopts consistent rules for both operational and stream interfaces. The ODL interface inheritance rules are as follows:

General and signature

R22 An interface template can be derived from one or several other interface templates, each of which is called a *base interface* (of 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.

R23 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.)

R24 A derived interface template may declare new sub-components (data types, operations or flows). Unless redefined, the sub-components of the base interface template can be referred to as if they were sub-components of the derived interface template. Interface inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope.

R25 It is illegal to inherit from two interface templates having the same operation identifiers[23], or having the same flow identifiers, or same data type identifiers.

R26 It is illegal to redefine an operation in the derived interface template[24].

R27 It is illegal to redefine an flow in the derived interface template.

R28 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.

Behavior

R29 The behaviorText of base interfaces are not available in a derived interface.

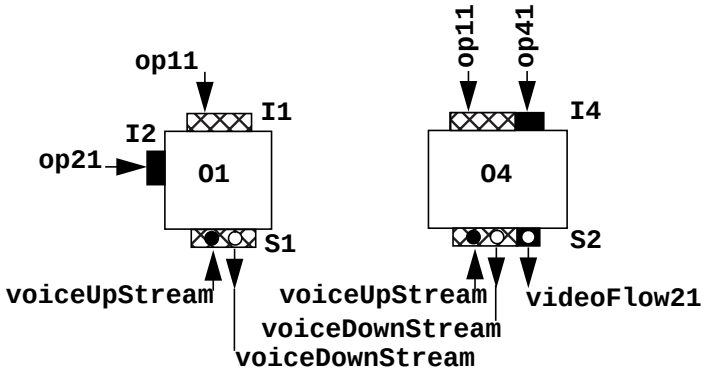
R30 The usage attribute of base interfaces are not available in a derived interface.

Trading attributes (see Section 3.8)

R31 The trading attributes on a derived interface is the union of the trading attributes on all of the base interfaces, plus any additional trading attributes specific to the derived interface.

As an illustrative example (see diagram below) assume that object `O4` declares as supported:

- interface `I4`, a specialization of interface `I1` constructed by the addition of `operation41`, and
- interface `S2`, a specialization of interface `S1`, with the addition of source `videoFlow21`.



It is possible to declare interface `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`:

Following are the ODL definitions of `I1 and S1`.

23. This is directly derived from OMG-IDL. A new version of OMG-IDL might relax this rule. The effect of this on TINA ODL is for further study.

24. This is directly derived from OMG-IDL. A new version of OMG-IDL might relax this rule. The effect of this on TINA ODL is for further study.

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

    // QoS type
    // max time server allowed for operation completion
    typedef … ServerResponseTime;

    void operation11 (in DataType11 …, out DataType12 …)
       with ServerResponseTime opResponseTime11;

}; // end of I1
```

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

    // QoS type
    typedef … VoiceQosType;

    source VoiceFlowType voiceDownStream
       with VoiceQosType voiceDownStreamQos;
    sink VoiceFlowType voiceUpStream
       with VoiceQosType voiceUpStreamQos;

}; // end of S1
```

The inheriting interfaces 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
       with FlowQosType11 flowQos21;
}; // end of S2
```

Object O4, using the inherited specialized interfaces, can be defined as follows:

```
object O4{
   behavior
   ...
   supports
      I4, S2;
   ...
}; // end of O4
```

3.7.3.2 Object Inheritance

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

- *Addition of interfaces*: new interfaces may be added to the list of supported interfaces of the base object.
- *Refinement of interfaces*: supported interfaces on the base objects may be specialized in the derived object.

The inheritance rules for objects are as follows:

General and supported interfaces

R32 An object template can be derived from one or several other object templates, each called a *base object* (or 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.

Remark: The inheritance tree for object templates is completely separated from the inheritance tree for interface templates.

R33 A derived object template may declare new sub-components (data types, interface templates). Unless redefined, the sub-components of the base object template can be referred to as if they were sub-components of the derived object template. Object inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope.

R34 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).

R35 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.

Remark: To add a new interface 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).

Remark: 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). The object may then offer instances of either of these interface templates.

R36 It is illegal to inherit from two object templates with the same interface template identifier declared separately in the scope of the two object templates.

Remark: If two interface templates defined in two object templates have the same identifier but are declared in the two different places (i.e., the interface declaration is not shared between the two objects), they are considered different.

R37 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.

Behavior

R38 The behaviorText of base objects are not available in a derived object.

R39 The required interfaces of the base object is the union of the required interfaces of the base objects, plus any additional required interfaces specific to the derived object.

Initial

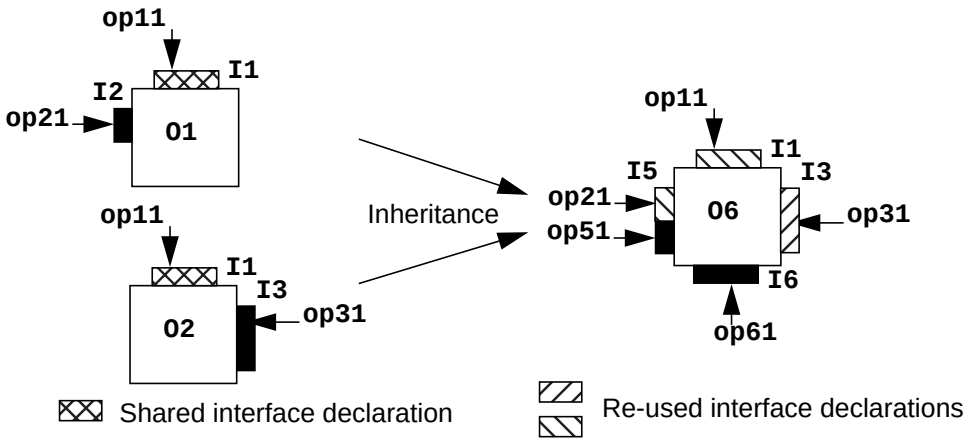
R40 The initial interfaces of base objects are not available in a derived object; initial interfaces are not inherited.

Note that the initial interface type of a derived object must be derived from (or may be identical to, in the case of single object inheritance) the initial interface types of direct base object types. Otherwise, a management system (for instantiation) will have difficulty seeing the derived object type as equivalent to it's base types.

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

- interface `I1` identical to the interface `I1` of `O1` and `O2` (3.3.2.4),
- interface `I5` supporting operation `operation21` as defined on interface `I2` of `O1` (3.3.2.4), and in addition `operation51`,
- interface `I3` similar to the interface `I3` of `O2` (3.3.2.4),
- interface `I6`, supporting operation `operation61`.

The following inheritance relationships can be established between objects `O1`, `O2`, and `O6`:



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 entities.

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

```
interface I5: I2{
   ...
   typedef … DataType51;
   void operation51(in DataType51 …);
}; // end of I5

interface I6{
   ...
   typedef … DataType61;
   void operation61(in DataType61 …);
}; // end of I6
```

Object O6 is declared as derived from objects O1 and O2, and interfaces I5 and I6 are declared in the list of supported interfaces of O6:

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

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

Since there is a relationship of inheritance from interface `I2` to interface `I5`, the interface `I2 from O2` is not considered as supported on `O6`, and instead interface `I5`, the refinement of `O2` is considered as supported on `O6`. If I2 was to be supported it would have to be explicitly listed as being supported.

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

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

3.7.3.3 Object Group Inheritance

It is assumed that group inheritance provides “re-use by specialization” of (an ODL specification of) a group. The group that is inherited from is called a *base group*. the group that does the inheriting is called the derived group. This specialization can take either of two basic forms:

- *Addition of objects/groups*: new objects may be added to the list of component objects of the base groups.
- *Refinement of objects/groups*: component objects/groups on the base groups may be specialized in the derived group.

The inheritance rules for groups are as follows:

General and component objects

R41 A group template can be derived from one or several other group templates, each called a *base group* (or base group 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.

Remark: The inheritance tree for object templates is completely separated from the inheritance trees for object and interface templates.

R42 A derived group template may declare new sub-components (data types, interface templates, object templates). Unless redefined, the sub-components of the base group template can be referred to as if they were sub-components of the derived group template. Group inheritance causes all identifiers in the closure of the inheritance tree to be imported into the current naming scope.

R43 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).

R44 The object/group templates which may comprise a derived group template is the union of the component object/group templates (declared) supported on all of the base group templates, plus any additional object/group templates declared supported on the derived group template.

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

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

R45 It is illegal to inherit from two group templates with the same object template identifier declared separately in the scope of the two group templates.

Remark: If two object templates defined in two group templates have the same identifier but are declared in two different places (i.e., the object declaration is not shared between the two groups), they are considered different.

R46 It is illegal to inherit from two group templates with the same interface template identifier declared separately in the scope of the two group templates.

Remark: If two interface templates defined in two group templates have the same identifier but are declared in two different places (i.e., the interface declaration is not shared between the two groups), they are considered different.

R47 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.

Behavior

R48 The behaviorText of base groups are not available in a derived group.

Initial

R49 The manager object of base groups are not available in a derived group.

Note that the initial object type of a derived group template must be derived from (or may be identical to, in the case of single object group inheritance) the initial objet types of direct base group types. Otherwise, the management system (for instantiation) is unlikely to see the derived object group type as equivalent to it's base types.

Contracts

R50 The contracts which comprise a derived group template is the union of the contracts comprising the base group templates, plus any additional contracts declared supported on the derived group template.

3.7.4 Naming and Scoping with respect to Inheritance

The following scoping rules are added to support inheritance capabilities.

R51 Inheritance introduces identifiers into the derived interface template, object template or object group template.

R52 Inheritance of interface templates, object templates or object group templates introduces multiple global ODL identifiers for the inherited identifiers.

R53 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, object or interface) inherited into the scope.The identifier is not searched for in enclosing scopes.

3.8 Trading

The DPE component of the TINA architecture supports the notion of a Trading service [24]. Entities with server interfaces can advertise their interface references at the Trading service. Potential clients of server interfaces can perform lookup operations at the Trading service. Typically, it is expected that clients base their search on one or more of:

1. *interface reference type* (eg. return all interfaces of type CsmConfiguration),
2. *context*, a logical, hierarchical partitioning of the search space (eg. return all interfaces of type CsmConfiguration in the context someTelcoAdminDomain)
3. *attribute value pairs* (eg. return all interfaces of type CsmConfiguration in the context someTelcoAdminDomain with an *admissionPolicy of someTelcoAuthorizationLevel*)

The interface reference type is intrinsic to the interface, and the value of context is generally specified at run time, as are attribute values. From the application developer's point of view one can think of context as a "well known" attribute value, while from the Trading service developer's point of view it facilitates more efficient storage and retrieval of interface references. In general, attribute types and values are specified on a per server instance basis. ODL supports the specification of general trading attribute types and variable identifiers. These attributes are intended to be utilized in application code when importing and exporting interfaces to the trader. Such attributes and variables are specified on a per interface template basis. It is expected that trading attribute variable identifiers, as specified in the interface templates, will be visible to application developers. In addition, these will be assigned values before being passed to the Trading service.

Trading attributes are specified in the scope of interfaces, and take the form of a "trAttribute" keyword followed by the string keyword and variable identifier. All trAttributes are of type string. The following example shows how trading attributes can be specified:

```
interface S1{
    ...

    // Trading attribute specification for interface
    trAttribute string owner;
    trAttribute string colorCapability;

    // flow types
    typedef … VoiceFlowType;

    // QoS type
    typedef … VoiceQosType;
```

```
    source VoiceFlowType voiceDownStream
       with VoiceQosType voiceDownStreamQos;
    sink VoiceFlowType voiceUpStream
       with VoiceQosType voiceUpStreamQos;

 }; // end of S1
```

The interface has two trading attributes, owner (name of the entity that owns the display) and colorCapability (whether the display associated with the sink flow is color or monochrome), of type string. The values of these attributes are expected to be assigned by the application developer before the interface reference is published in the trading service.

4. Syntax of ODL

This section defines the syntax of ODL. It is divided into four parts which deal with the major components of ODL: type and constant declaration; interface templates; object templates; object group templates.

Note that example code segments in this document use lexical conventions, and include preprocessor directives (aka. compiler directives) in addition to ODL language statements. lexical conventions follow those of OMG IDL ([37], section 3.2). For example, "//" is frequently used to indicate comments, but is not a part of ODL. The preprocessor directives also follow those of OMG IDL ([37], section 3.3), which in turn follow those of ANSI C++. It should be noted that ODL is independent of language mappings, and the use of compiler directives from a C++ mapping are incidental to the description of ODL presented here.

4.1 Type and Constant Declaration

4.1.1 Structure

Data types and constants, can be declared in almost any scope within an ODL specification. These types or constants can be used for declaration of operation, exception, flow, and other template components. 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.

4.1.1.1 ODL Syntax for Type and Constant Declarations

The syntax supported by ODL for type and constant declaration is identical to the one of OMG-IDL. The reader can find in Appendix A a description of this syntax. Rules 68 to 81 define constant declarations, while rules 82 to 120 define type declarations.

4.1.1.2 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:
         struct Interval {
```

```
            Bps min;
            Bps maxd;
         };
   };
   union Jitter switch (Guarantee) {
      case Statistical: Bps mean;
      case Deterministic: Bps peak;
   };
};
```

4.2 Interface Template

4.2.1 Structure

The reader is assumed to have a knowledge of the interface architecture [5]. A computational interface template comprises:

- A behavior specification,
- A trading attribute specification,
- As appropriate, either:
  - an operational interface signature,
  - a stream interface signature.

This structure is reflected in the ODL rule for <interface_body> (see following sub-section). The subsequent sections of this document examine the elements of this structure in more detail.

4.2.1.1 ODL Syntax for Interface Template Declaration

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_behavior_spec>]
[<trading_attributes_spec>]
{ <op_sig_defns> | <stream_sig_defns> }

<interf_behavior_spec> ::= “**behavior**” {
{<interf_behavior_text> [<interf_usage_spec>]}
| <interf_usage_spec> }

<interf_behavior_text> ::=“**behaviorText**” <string_literal> “;”

<interf_usage_spec> ::=“**usage**” <string_literal> “;”

<trading_attributes_spec> ::= { <trading_attribute_spec> ";" }*

<trading_attribute_spec> ::= "**trAttribute**" "**string**" <trading_attr_name>

<trading_attr_name>::= <simple_declarator>

4.2.1.2 Example of Interface Template Declaration

Below is an example of an interface `CSMConfiguration` that is derived by inheritance from an interface `ServiceManagement` as defined in the module `Management`.

```
interface CSMConfiguration: Management::ServiceManagement{
   ...
} // end CSMConfiguration
```

4.2.2 Interface Behavior Specification

The interface signature describes only the syntactic structure of an interface. Signature compatibility is less discerning than behavior compatibility. It is indeed possible that two interfaces have compatible signatures but differ completely in their behavior. This section describes how behavior is specified in ODL.

4.2.2.1 ODL Syntax for Interface Behavior and Usage Specification

The following syntax is defined for the interface behavior specification:

<interf_behavior_spec> ::= "**behavior**" {
{<interf_behavior_text> [<interf_usage_spec>]}
| <interf_usage_spec> }

<interf_behavior_text> ::="**behaviorText**" <string_literal> ";"

<interf_usage_spec> ::="**usage**" <string_literal> ";"

4.2.2.2 Example of Interface Behavior Specification

Below is an example of an interface `CSMConfiguration`, which includes a behavior specification comprising behavior text as well as a usage specification.

```
interface CSMConfiguration: Management::ServiceManagement {

   behavior
      behaviorText
      "This interface serves to configure the object 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.

        Concurrent calls to ReadState operations are permitted but
        all operation invocations are blocked while a WriteState
        operation is being handled.

        WriteState operations will block until all current
        ReadState or WriteState invocations are completed.";
    ...
}; // end of CSMConfiguration
```

4.2.3 Trading Attributes

"Trading attributes" describe properties of an interface used in constraint specifications when trading or for particular interface references. For each interface type (whether an operational or stream interface), it is possible to specify a list of parametrized qualities, each of which is associated with an operation or stream interface. These quantities are typically specified by a server (or another object acting on behalf of the server) and used when exporting an interface reference to the trader. A client of that interface type may express its requirements to the trader using this set of parameters.

4.2.3.1 ODL syntax for Interface Trading Attributes

The following syntax is defined for trading attribute specifications:

<trading_attributes_spec> ::= { <trading_attribute_spec> ";" }*

<trading_attribute_spec> ::= "**trAttribute**" "**string**" <trading_attr_name>

<trading_attr_name>::= <simple_declarator>

4.2.3.2 Example of Interface Trading Attribute

The following example shows the use of an interface attribute declaration.

Below is an example of an interface `CSMConfiguration`, which includes trading attribute specification.

```
interface CSMConfiguration: Management::ServiceManagement {

    behavior
        behaviorText
        "This interface serves to configure the object 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.

      Concurrent calls to ReadState operations are permitted but
      all operation invocations are blocked while a WriteState
      operation is being handled.

      WriteState operations will block until all current
      ReadState or WriteState invocations are completed.";

   // Trading attribute specification
   typedef enum { easyAccess, secureAccess } AccesType ;

   trAttribute string owner;
   trAttribute string SecurityKeyLength;

   ...
} // end of CSMConfiguration
```

4.2.4 Operational interface signature

An operational interface signature comprises a set of interrogation and announcement signatures, one for each operation type in the interface. An operational interface signature specifies the following information (similar to OMG-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 operation quality of service parameter
- An optional "raises" expression which indicates which exceptions may be raised as a result of an invocation of this operation.

Support similar to OMG-IDL's attribute declaration is provided to simplify the specification of get and set operations.[1]

1. This should not be confused with "Trader Attributes."

4.2.4.1 ODL syntax for Operational Interface Signature

The following syntax is defined for the signature of operational interfaces. It is similar to the OMG-IDL syntax for interface template declaration. Note that the operational signature definition is extended with service attribute definitions to enable the expression of QoS constraints on each operation.

<op_sig_defns> ::= { <op_sig_defn> ";" }$^{+}$

<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>

<announcement> ::= "**oneway**" "**void**" <identifier> <parameter_dcls>

<interrogation> ::=<attr_dcl>
| <oper_dcl>

<attr_dcl> ::= ["**readonly**"] "**attribute**" <simple_type_spec>
<declarators>

<oper_dcl> ::= <op_type_spec> <identifier>
<parameter_dcls>
[<raises_expr>] [<context_expr>]

<op_type_spec> ::=<simple_type_spec>
| "**void**"

<parameter_dcls> ::= "(" <param_dcl> { "," <param_dcl> }* ")"
| "(" ")"

<param_dcl> ::=<param_attribute> <simple_type_spec>
<declarator>

<param_attribute> ::= "**in**"
| "**out**"
| "**inout**"

<raises_expr> ::= "**raises**"
"(" <scoped_name> { "," <scoped_name> }* ")"

<context_expr> ::= "**context**"
"(" <string_literal> { "," <string_literal> }* ")"

4.2.4.2 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. These attribute declarations should not be confused with the trading or quality of service attributes defined above.

4.2.4.3 Example of Operational Interface Signature

Below is an example of an interface, `Trail`. It has three interface attributes (`nml_cp`, `neighbors`, `state`) and five operations (`create_trail`, `destroy_trail`, `modify_traffic_description`, `modify_qos`, `modify_trail_policies`).

```
interface Trail {
    ...
    attribute CP::SNC nml_cp;
    attribute LncNeighborList neighbors;
    attribute ManagementState state;

    void create_trail (
              in TTPoint origin,
              in TTPlist destination,
              in TrailDescription desc,
              in TrailPolicies policies,
              out TrailId trail);
    void destroy_trail (in TrailId trail);
    void modify_traffic_description(
              in TrailId trail,
              in PacketFlowTrafficDescription trafficDescr);
    void modify_qos(in TrailId trail, in PacketFlowQos qos);
    void modify_trail_policies(
              in TrailId trail,
              in TrailPolicies policies);
    };
```

4.2.5 Stream (Flow) Signature

A stream interface 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.

It should be noted that 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 as shown in Figure 4-1. Note that the server type includes a declaration that it “supports” the stream interface, while the client type includes a declaration that it “requires” the stream interface.

For example, in order to play a video game, a client (the `Player`) locates an appropriate interface (`VideoGame`) to the server (the `Game`). The service is defined naturally in terms of information sources (the `picture` and `sound`) 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 specification, as shown in Figure 4-1.

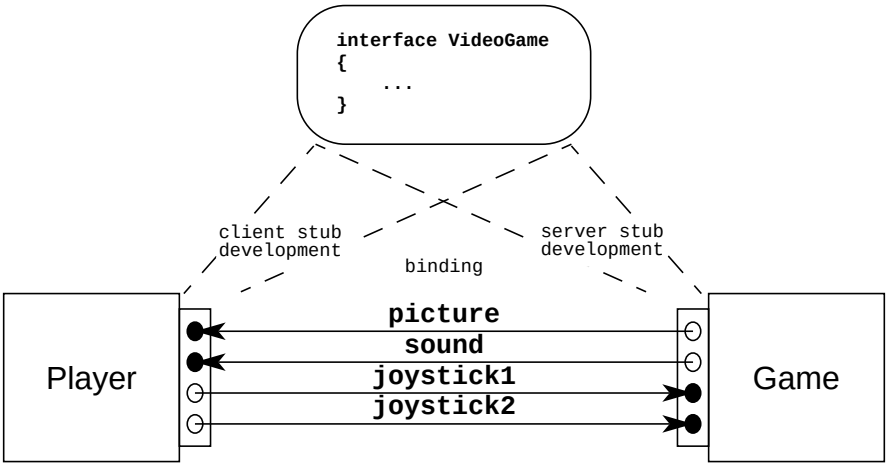


Figure 4-1. Example of stream interface template usage.

In ODL, stream interfaces 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 definition of the server, the stream interface is listed as a supported interface, while on the client, the interface is listed as a required interface.

4.2.5.1 ODL syntax for Stream (Flow) Signature Declaration

<stream_flow_defns>::=<flow_direction> <flow_type> <identifier>
["**with**" <QoS_attribute>]

<flow_direction> ::= "**source**" | "**sink**"

<flow_type> ::= <type_spec>

4.2.5.2 Example of Flow Signature Declaration

Following is an example of a stream interface, `gameServer`, with 4 flows (`joystick, joystick2, picture, sound`).

```
interface gameServer {
   ...
   sink JoystickFlowType joystick1 with JoystickQos joystick1Qos;
```

```
   sink JoystickFlowType joystick2 with JoystickQos joystick2Qos;
   source VideoFlowType  picture  with VideoQos pictureQos;
   source AudioFlowType  sound    with AudioQos soundQos;
};
```

4.2.6 Interface Inheritance

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

4.2.6.1 ODL Syntax for Interface Inheritance

The following syntax is defined for interface inheritance:

<interface_header> ::= “**interface**” <identifier> [<interf_inheritance_spec>]

<interf_inheritance_spec> ::= “:” <scoped_name> { “,” <scoped_name> }*

4.2.6.2 OMG Implications

It should be noted that the OMG-IDL specification, in its current form [37], prohibits redefinition of an operation identifier in a derived interface type specification and prohibits inheritance of two operations of the same identifier. Initially, ODL will be restricted according to this limitation of OMG-IDL in operational and stream interface definitions.

4.2.6.3 Use of Inheritance

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

It should be noted that a stream interface cannot inherit from an operational interface and vice versa.

4.2.6.4 Example of Interface Inheritance

Following is an example of an interface `CSMConfiguration` that is derived by inheritance from an interface `ServiceManagement` that is defined in the module `Management`.

```
module Management{
   interface ServiceManagement{
         behavior
```

```
            behaviorText
            “This allows management of a service....”
      ...
      attribute short edge_nb;
      oneway create_edge (in EdgeRef leaf);
      void attach_edge (in EdgeId leaf)
            with BoundedResponseTime attach_edgeQos;
      ...
   }; // end of interface ServiceManagement
}; // end of module Management


interface CSMConfiguration: Management::ServiceManagement{
   behavior
      behaviorText
      “This allows configuration of the CSM object...”
   ...
}; // end of interface CSMConfiguration
```

The specification of the `CSMConfiguration` interface in this example is consistent with the following example specification:

```
interface CSMConfiguration{
   behavior
      behaviorText
      “This allows management of a service....
      This allows configuration of the CSM object...”
   ...
   attribute short edge_nb;
   oneway create_edge (in EdgeRef leaf);
   void attach_edge (in EdgeId leaf)
         with BoundedResponseTime attach_edgeQos;
   ...   // definitions specific to ServiceManagement
   ...   // additional definitions specific to CSMConfiguration
}; // end of interface CSMConfiguration
```

4.3 Object Template

4.3.1 Structure

The reader is assumed to have a knowledge of the object architecture as presented in [5]. An object template specification comprises two high level parts. The first supports inheritance, and is associated with the declaration of the object’s identifier. The second part is the object body, which comprises the main sub-parts of the template as follows:

- A behavior specification,
- A specification of supported interfaces.
- An initialization specification,

Below we will examine these specifications in more detail, as well as the syntax that supports inheritance.

4.3.1.1 ODL Syntax for Object Template Declaration

The following syntax is defined for object template declaration:

<object_template> ::=<object_template_header>
"{" <object_template_body> "}"

<object_template_header> ::= "**object**" <identifier>
[<object_inheritance_spec>]

<object_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*

<object_template_body> ::= [<object_behavior_spec>]
<suptd_interf_templates>
[<object_init_spec>]

<object_behavior_spec> ::="**behavior**" {
{<object_behavior_text> [<reqrd_interf_templates>] }
| <reqrd_interf_templates> }

<object_behavior_text> ::="**behaviorText**" <string_literal> ";"

<reqrd_interf_templates> ::= "**requires**" <req_interf_defn>
{"," <req_interf_defn> }* ";"

<req_interf_defn> ::= <scoped_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> ";"

4.3.1.2 Example of Object Template Declaration

Following is an example showing how an object template is declared. It begins with the keyword "object", which is then followed by the identifier of the object type, `acmeCsmFactory`. This template doesn't inherit from any other, as indicated by the absence of any inheritance specifications. The body of the template is then declared between the braces.

```
object AcmeCsmFactory {
    …
};
```

4.3.2 Object Behavior Specification

The behavior of an object is specified in two parts. The first is a text string that should describe the role of an object in providing services via each of its interfaces. The second comprises the (declared) required interfaces which specifies interface types used by instances of the object to perform their functions and provide their services.

4.3.2.1 ODL Syntax for Object Behavior Template

The following syntax is defined for object behavior specification:

<object_behavior_spec> ::= “**behavior**” {
{<object_behavior_text> [<reqrd_interf_templates>] }
| <reqrd_interf_templates> }

<object_behavior_text> ::=“**behaviorText**” <string_literal> “;”

<reqrd_interf_templates> ::= “**requires**” <req_interf_defn>
{“,” <req_interf_defn> }* “;”

<req_interf_defn> ::=<scoped_name> | <interface_template>

4.3.2.2 Example of Object Behavior Declaration

Following is an example of the use of the behavior specification. A `Timer` object, calls an operation `tick` on the `TimeInterrupt` interface of another object.

```
object Timer{

   behavior
      behaviorText
      “Instances of this object periodically call the
      tick function of a specified TimerInterrupt
      interface.”;

      requires
         TimeInterrupt;
   ...
};
```

4.3.3 Supported Interfaces

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

4.3.3.1 ODL Syntax for an Object's Supported Interfaces

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>

4.3.3.2 Example of Object Supported Interfaces

Below is an example which shows the use of the supported interfaces specification.

```
object Timer{

   behavior
      behaviorText
      "Instances of this object periodically call the
      tick function of a specified TimerInterrupt
      interface.";

      requires
         TimeInterrupt;

   supports
      PeriodicMgmt,
      TimerService;
   ...
};
```

4.3.4 Object 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 must also be one of the supported interfaces (see below).

4.3.4.1 ODL Syntax for Object Initialization Specification

The following syntax is defined for the initialization specification:

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

4.3.4.2 Example of Object Initialization Specification Declaration

Below is an example of the use of the object initialization specification. It extends from the previous example of a `Timer` object. An instance of the initial interface template, `PeriodicMgmt`, is returned to the creator of the object. This interface is used to manage the object instance.

```
object Timer{

   behavior
      behaviorText
      "Instances of this object periodically call the
      tick function of a specified TimerInterrupt
      interface.";

      requires
         TimeInterrupt;

   supports
      PeriodicMgmt,
      TimerService;

   initial
      PeriodicMgmt;
};
```

4.3.5 Object Inheritance

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

4.3.5.1 ODL Syntax for Object Inheritance

The following syntax is defined for object inheritance:

<object_template_header> ::= "**object**" <identifier>[<object_inheritance_spec>]

<object_inheritance_spec> ::= "**:**" <scoped_name> { "**,**" <scoped_name> }*

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

There are no restrictions on the names of interfaces or types inherited from 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 types of the corresponding base object classes.

In the case of required and supported interface types, the derived object type can be considered to require or support the “restricted set” of interface types as this set is defined in Section 3.7.

4.3.5.2 Example of Object Inheritance.

Below is an example which shows an object `MyCSMfactory` derived from the object `CSM-factory`:

```
object CSMfactory {

    behavior
        requires
            QoSmanagerIF;

    supports
        Management,
        LcgFactory,
        CSMConfiguration;

    initial
        Management;

}; // end CSMfactory


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


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


object MyCSMfactory: CSMfactory{

    behavior
        requires
            AccountingEventIF;

    supports
        MyManagement,
        MyCSMConfiguration;

    initial
```

```
        MyManagement;

};  // end MyCSMfactory
```

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

4.4 Object Group template

4.4.1 Structure

The reader is assumed to have a knowledge of the object group architecture [5], [23]. A group template specification comprises two high level parts. The first supports inheritance, and is associated with the declaration of the group's identifier. The second part is the group body, which comprises the main sub-parts of the template as follows:

- A behavior specification,
- A specification of supported objects and object groups.
- An initialization specification,
- A specification of interfaces visible outside the group,

Below we will examine these specifications in more detail, as well as the syntax that supports inheritance.

4.4.1.1 ODL Syntax for Group Template Declaration

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_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*

<group_template_body> ::= [<group_behavior_spec>]
<supp_comp_templates>
[<group_init_spec>]
[<contract_interfaces>]

<group_behavior_spec> ::= "**behavior**" <group_behavior_text>

<group_behavior_text> ::="**behaviorText**" <string_literal> ";"

<supp_comp_templates>::= “**components**” <suptd_comp> “;”

<suptd_comp> ::=<suptd_comp_defn> {“,” <suptd_comp_defn> }*

<suptd_comp_defn> ::= <scoped_name>

<group_init_spec> ::= “**manager**” <scoped_name> “;”

<contract_interfaces> ::= “**contracts**” <scoped_name> { “,” <scoped_name> }* “;”

4.4.1.2 Example of Group Template Declaration

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

```
group subnetManager {
    ...
};
```

4.4.2 Object Group Behavior and Initialization Specification

The behavior specifications of an object group strongly parallels that of an object. The initialization specification of an object group is slightly different to that of an object in that it specifies an object template to be instantiated instead of an interface template. In both cases an interface template is expected to be returned to the creator. It should be noted that the manager object must be one of the component objects and the returned interface, one of the contracts (see below).

4.4.2.1 ODL Syntax for Group Behavior and Initialization Declaration

The following syntax is defined for group behavior and initialization specification:

<group_behavior_spec> ::= “**behavior**” <group_behavior_text>

<group_behavior_text> ::= “**behaviorText**” <string_literal> “;”

<group_init_spec> ::= “**manager**” <scoped_name> “;”

4.4.2.2 Example of Group Behavior and Initialization Declaration

Following is an example of an object group specification for the group `subnetManager`.

```
interface Configuration {...};

interface Configurator {...};
```

```
object CMC {
    behavior
        requires
            Configuration;
    supports
        Configurator;
    ...
};


group SubnetManager {

    behavior
        behaviorText
        "This group manages a subnetwork"

    ...

    manager
        CMC;
    ...
};
```

4.4.3 Contracts

Contracts are the interfaces of the object group that are visible to entities outside the object group. In the object group specification these interfaces are indicated as a simple list of (optionally scoped) identifiers.

4.4.3.1 ODL Syntax for Contract Declaration

Following is the syntax supporting contracts.

<contract_interfaces> ::= "**contracts**" <scoped_name> { "," <scoped_name> }* ";"

4.4.3.2 Example of Contract Declaration

Below we extend the previous example to include contracts.

```
interface Configuration {...};


interface Configurator {...};
```

```
interface Trail {...};


interface TC {...};


object CMC {
   behavior
      requires
         Configuration;
   supports
      Configurator;
   ...
};


group SubnetManager {

   behavior
      behaviorText
      “This group manages a subnetwork”

   ...

   manager
      CMC;

   contracts
      Configurator,
      Trail,
      TC;
};
```

4.4.4 Component Objects and Groups

Component objects and object groups are the object and object group templates that can be instantiated within the object group.

4.4.4.1 ODL Syntax for Component Declaration

Following is the syntax supporting component objects.

<supp_comp_templates>::=“**components**” <suptd_comp> “;”

<suptd_comp> ::=<suptd_comp_defn> {“,” <suptd_comp_defn> }*

<suptd_comp_defn> ::= <scoped_name>
| <object_template>
| <group_template>

4.4.4.2 Example of Component Declaration

Below we extend the previous example to include contracts.

```
interface Configuration {...};


interface Configurator {...};


interface Trail {...};


interface TC {...};


object CMC {
   behavior
      requires
         Configuration;
   supports
      Configurator;
   ...
};


object NetworkCoordinator {
   behavior
      requires
         TC, SncService, SncServiceFactory;
   supports
      Trail, TC, Configuration;
   ...
};


object NetworkCP {
   supports
      SncService, SncServiceFactory, Configuration;
   ...
};


object ElementCP {...};


group SubnetManager {

   behavior
      behaviorText
```

```
        "This group manages a subnetwork"

    components
        CMC, NetworkCoordinator, NetworkCP, ElementCP;

    manager
        CMC;

    contracts
        Configurator, Trail, TC;

};
```

4.4.5 Object Group Inheritance

Object group 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 specifications is to support the use of object framework specifications.

4.4.5.1 ODL Syntax for Group Inheritance Declaration

The following syntax is defined for group inheritance:

<group_template_header> ::= "**group**" <identifier> [<group_inheritance_spec>]

<group_inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }*

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

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

The manager object specified in a derived object template must be of a type which is derived from all manager object types of the corresponding base object classes.

5. How to use ODL

This chapter discusses ways in which ODL specifications can be used.

5.1 ODL Tool

This section describes one use of ODL specifications, and their position in the overall process of application development (see Figure 5-1.).

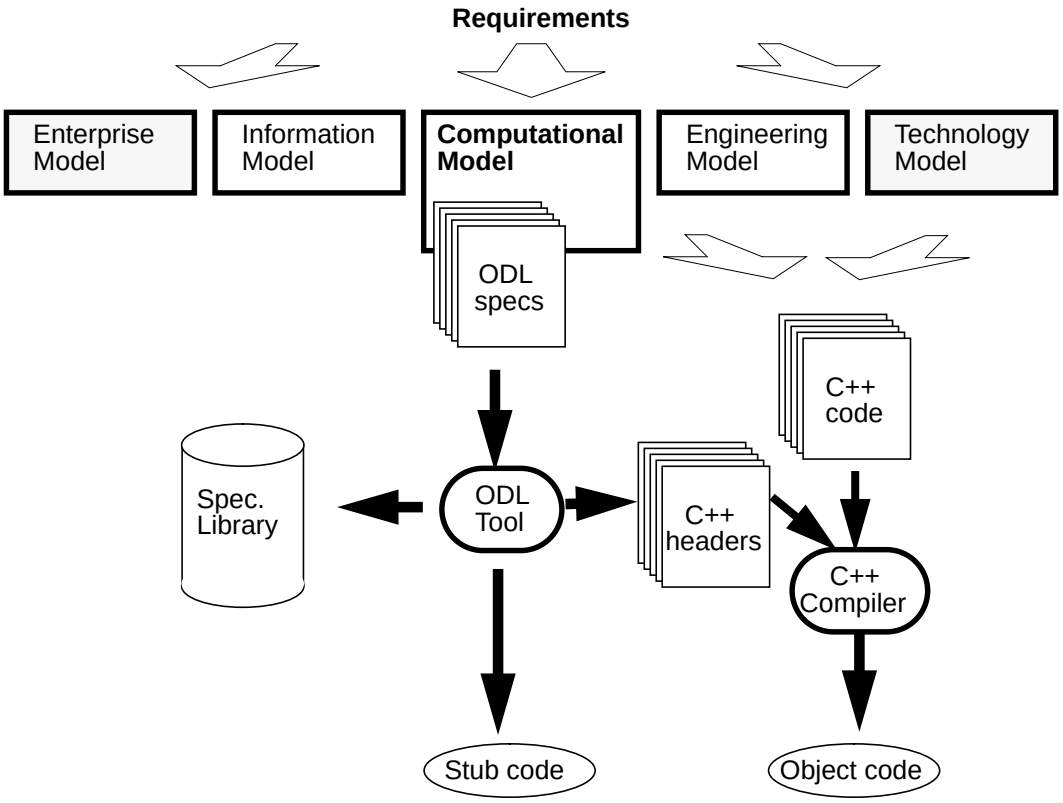


Figure 5-1. Use of ODL specifications in a C++ development environment

Following the ODP approach, TINA applications are viewed from 5 viewpoints (enterprise, information, computational, engineering, and technology).[1] In principle a language for each of the viewpoints can be specified. ODL has been developed to enable the computational viewpoint of an application to be described. There are a number of ways that a language such as ODL (and languages pertinent to other viewpoints) can be used in a software engineering methodology. For example the enterprise viewpoint language can take on the role of the requirements specification language, and include policies on issues like accounting and security. The information viewpoint language can be used to represent the conceptual domain of the intended system. The Engineering viewpoint language may be used to spec-

1. TINA defines modelling concepts for the information, computational, and engineering viewpoints.

ify configuration and distribution information. The technology viewpoint language may be used to specify particular technology information underlying the system. The computational viewpoint language may be used to drive the bulk of the programming language implementation of the system. The entities that appear in the computational viewpoint language may be derived from those in the information viewpoint language, and be influenced by the specifications in the enterprise and engineering viewpoint languages.

Ultimately the system specification in ODL needs to be mapped to a programming language, such as C++, for implementation of the application. Languages other than C++ can be used to implement ODL specified systems. However, for illustrative purposes, the following compilation chain deals with the processing ODL specification with a C++ language mapping and C++ application files.

Once the system specifications have been written in ODL, a tool called *ODL tool*, processes them. It produces three types of outputs:

- Object group, object, and interface descriptions, that can be stored in a repository/library, for future re-use.
- Stub code for the interfaces described (in a form that is compilable, see [9]).
- Text files (for example in the form of C++ header files) enabling the linking of the stub codes with object implementation code.

ODL tool can operate by initially passing input through a standard C++ preprocessor, supporting macro expansion and file inclusion. Preprocessing may or may not be performed by a distinct application.

A C++ compiler can then take as input the C++ description of the application implementation, and the C++ header files, in order to produce executable code.

As stated in section 3, interfaces objects and groups can be specified independently. One possible way in which ODL specifications can be documented is to separate interface, object and group declarations into three different kinds of files. These three different source files can be used at compile time as input for the ODL tool (see Figure 5-2.).

This approach offers several advantages [28], the main two being:

- Clear separation between group, object and interface declaration;
- Possibility to re-use directly OMG-IDL specifications.

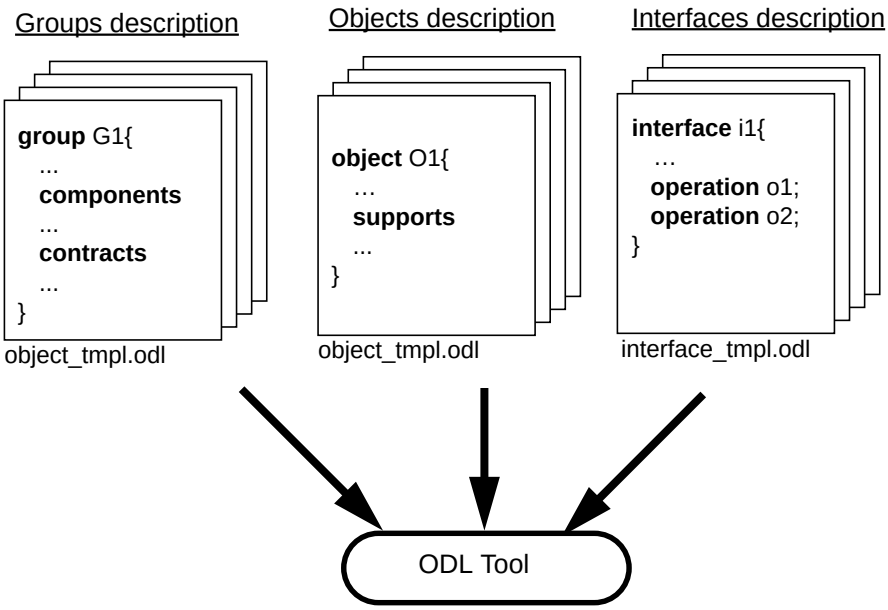


Figure 5-2. Inputs to ODL tool.

6. Acknowledgments

The editor acknowledges the valuable cumulative input of review comments, prior editorships and authorships, and contributions from related documents received from:

- Martin Chapman (BT)
- Heine Christensen (Tele Danmark)
- Eric Colbern (Telenor)
- Fabrice Dupuy (CNET)
- Frederic Dang-Tran (CNET)
- Ennio Grasso (CSELT)
- Takao Hamada (Fujitsu)
- Tom Handegard (Telenor)
- Nigel Hooke (Telstra)
- Mikael Jorgensen (Tele Danmark)
- Barry Kitson (Telstra)
- Peter Leydekkers (PTT Nederland)
- Henry Lockyer (Ericsson)
- Subrata Mazumdar (IBM)
- Nicholas Mercouroff (Acatel)
- Corrado Moiso (CSELT)
- Osamu Miyagishi (NTT)
- Chris Mugden (Telstra)
- Narayanan Natarajan (Bellcore)
- Ajeet Parhar (Telstra)
- Juan Pavón (Acatel)
- Stephane Pensivy (France Telecom)
- Fernando Ruano (Telefonica)
- Nikolaus Singer (Acatel)
- Vincent Stinesen (PTT Nederland)
- Joe Sventek (Hewlett-Packard)
- Paul Vickers (Hewlett-Packard)
- Geoff Wheeler (Telstra)

Ajeet Parhar.
Telstra
TINA-C Core Team

Appendix A. BNF description of ODL

A.1 ODL Lexical Conventions

The lexical and preprocessor conventions of OMG-IDL are assumed [37]. It should be noted that OMG-IDL extends the 52 alphabetic characters used on English keyboards using the ISO Latin-1 character set. The graphic characters are also unusually rich, extending the familiar 32 character set to a 65 character set. The decimal digits, formatting characters and escape sequences are not unusual.

A.2 ODL Keywords

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

any	**attribute**	**behavior**	**behavorText**
boolean	**case**	**char**	**components**
const	**context**	**contracts**	**default**
double	**enum**	**exception**	**FALSE**
float	**group**	**in**	**initial**
inout	**interface**	**long**	**manager**
module	**Object**	**object**	**octet**
oneway	**out**	**raises**	**requires**
readonly	**sequence**	**short**	**sink**
source	**string**	**struct**	**supports**
switch	**trAttribute**	**TRUE**	**typedef**
unsigned	**union**	**usage**	**void**
with			

A.3 ODL extended BNF Notation

The following meta-symbols are use to describe ODL's syntax. The description is an extended Backus Naur Form (BNF)[1].

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

1. 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}

{} the enclosed syntactic units are grouped as a single syntactic unit

[] the enclosed syntactic unit is optional - may occur zero or one time

A.4 ODL Syntax

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

A.4.1 Top Level Syntax

1. <odl_spec> ::= <definition>*
2. <definition> ::= <module> “;”
| <group_dcl> “;”
| <object_dcl> “;”
| <interface_dcl> “;”
| <supporting_def> “;”

A.4.2 Module Syntax

3.* <module> ::= “**module**” <identifier> “{“ <definition>$^{+}$ “}”

4.* <scoped_name> ::= <identifier>
| “::” <identifier>
| <scoped_name> “::” <identifier>

A.4.3 Group Syntax

5. <group_dcl> ::= <group_forward_dcl>
| <group_template>
6. <group_forward_dcl> ::= “**group**” <identifier>
7. <group_template> ::= <group_template_header>
“{“ <group_template_body> “}”
8. <group_template_header> ::= “**group**” <identifier>
[<group_inheritance_spec>]
9. <group_inheritance_spec> ::= “:” <scoped_name> { “,” <scoped_name> }*
10. <group_template_body> ::= [<supporting_def_spec>]
[<interface_def_spec>]
[<object_def_spec>]

[<group_behavior_spec>]
<supp_comp_templates>
[<group_init_spec>]
[<contract_interfaces>]
11. <supporting_def_spec> ::= {<supporting_def> “;”}*
12. <interface_def_spec> ::= {<interface_dcl> “;”}*
13. <object_def_spec> ::= {<object_dcl> “;” }*

14. <group_behavior_spec> ::= “**behavior**” <group_behavior_text>
15. <group_behavior_text> ::= “**behaviorText**” <string_literal> “;”
16. <supp_comp_templates> ::= “**components**” <supp_comp> “;”
17. <supp_comp> ::= <supp_comp_defn> {“,” <supp_comp_defn> }*
18. <supp_comp_defn> ::= <scoped_name>
19. <group_init_spec> ::= “**manager**” <scoped_name> “;”
20. <contract_interfaces> ::= “**contracts**” <scoped_name>
{ “,” <scoped_name> }* “;”

A.4.4 Object Syntax

21. <object_dcl> ::= <object_forward_dcl>
| <object_template>
22. <object_forward_dcl> ::= “**object**” <identifier>
23. <object_template> ::= <object_template_header>
“{“ <object_template_body> “}”
24. <object_template_header> ::= “**object**” <identifier>
[<object_inheritance_spec>]
25. <object_inheritance_spec> ::= “:” <scoped_name> { “,” <scoped_name> }*
26. <object_template_body> ::= [<supporting_def_spec>]
[<interface_def_spec>]

[<object_behavior_spec>]
<suptd_interf_templates>
[<object_init_spec>]
27. <object_behavior_spec> ::= “**behavior**” {
{ <object_behavior_text>
[<reqrd_interf_templates>] }
| <reqrd_interf_templates> }
28. <object_behavior_text> ::= “**behaviorText**” <string_literal> “;”
29. <reqrd_interf_templates> ::= “**requires**” <req_interf_defn>
{“,” <req_interf_defn> }* “;”
30. <req_interf_defn> ::= <scoped_name>
31. <suptd_interf_templates> ::= “**supports**” <suptd_interf> “;”
32. <suptd_interf> ::= <suptd_interf_defn> {“,” <suptd_interf_defn> }*
33. <suptd_interf_defn> ::= <scoped_name>
34. <object_init_spec> ::= “**initial**” <scoped_name> “;”

A.4.5 Interface Syntax

35. <interface_dcl> ::= <interface_forward_dcl>
| <interface_template>
36. <interface_forward_dcl> ::= “**interface**” <identifier>

37. <interface_template> ::= <interface_header> “{“ <interface_body> “}”

38. <interface_header> ::= “**interface**” <identifier>
[<interf_inheritance_spec>]

39. <interf_inheritance_spec> ::= “:” <scoped_name> { “,” <scoped_name> }*

40. <interface_body> ::= [<supporting_def_spec>]

[<interf_behavior_spec>]
[<trading_attributes_spec>]
{ <op_sig_defns> | <stream_sig_defns> }

41. <interf_behavior_spec> ::= “**behavior**” {
{<interf_behavior_text> [<interf_usage_spec>]}
| <interf_usage_spec> }

42. <interf_behavior_text> ::= “**behaviorText**” <string_literal> “;”

43. <interf_usage_spec> ::= “**usage**” <string_literal> “;”

44. <trading_attributes_spec> ::= { <trading_attribute_spec> “;” }*

45. <trading_attribute_spec> ::= “**trAttribute**” “**string**” <trading_attr_name>

46. <trading_attr_name> ::= <simple_declarator>

A.4.6 (Operational) Interface Syntax

47. <op_sig_defns> ::= { <op_sig_defn> “;” }*

48. <op_sig_defn> ::= { <announcement> | <interrogation> }
[**“with”** <QoS_attribute>]

49. <QoS_attribute> ::= <QoS_attr_type> <QoS_attr_name>

50. <QoS_attr_type> ::= <simple_type_spec>

51. <QoS_attr_name> ::= <simple_declarator>

52. <announcement> ::= “**oneway**” “**void**” <identifier> <parameter_dcls>

53. <interrogation> ::= <attr_dcl>
| <oper_dcl>

54. <attr_dcl> ::= [“**readonly**”] “**attribute**” <param_type_spec>
<declarators>

55. <oper_dcl> ::= <op_type_spec> <identifier>
<parameter_dcls>
[<raises_expr>] [<context_expr>]

56.*<op_type_spec> ::= <param_type_spec>
| “**void**”

57.*<parameter_dcls> ::= “(“ <param_dcl> { “,” <param_dcl> }* “)”
| “(“ “)”

58.*<param_dcl> ::= <param_attribute> <param_type_spec>
<declarator>

59.*<param_attribute> ::= “**in**” | “**out**” | “**inout**”

60.*<raises_expr> ::= "**raises**"
"(" <scoped_name> { "," <scoped_name> }* ")"

61.*<context_expr> ::= "**context**"
"(" <string_literal> { "," <string_literal> }* ")"

62.*<param_type_spec> ::= <base_type_spec>
| <string_type>
| <scoped_name>

A.4.7 (Stream) Interface Syntax

63. <stream_sig_defns> ::= { <stream_flow_defn> ";" }*

64. <stream_flow_defn> ::= <flow_direction> <flow_type>
<identifier> [**"with"** <QoS_attribute>]

65. <flow_direction> ::= "**source**" | "**sink**"

66. <flow_type> ::= <param_type_spec>

A.4.8 Supporting Definition Syntax

67. <supporting_def> ::= <const_dcl> ";"
| <type_dcl> ";"
| <except_dcl> ";"

68.*<const_dcl> ::= "**const**" <const_type>
<identifier> "=" <const_exp>

69.*<const_type> ::= <integer_type>
| <char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <scoped_name>

70.*<const_exp> ::= <or_expr>

71.*<or_expr> ::= <xor_expr>
| <or_expr> "|" <xor_expr>

72.*<xor_expr> ::= <and_expr>
| <xor_expr> "^" <and_expr>

73.*<and_expr> ::= <shift_expr>
| <and_expr> "&" <shift_expr>

74.*<shift_expr> ::= <add_expr>
| <shift_expr> ">>" <add_expr>
| <shift_expr> "<<" <add_expr>

75.*<add_expr> ::= <mult_expr>
| <add_expr> "+" <mult_expr>
| <add_expr> "-" <mult_expr>

76.*<mult_expr> ::= <unary_expr>
| <mult_expr> "*" <unary_expr>
| <mult_expr> "/" <unary_expr>
| <mult_expr> "%" <unary_expr>

77.*<unary_expr> ::= <unary_operator> <primary_expr>
| <primary_expr>

78.*<unary_operator> ::= “-”
| “+”
| “~”

79.*<primary_expr> ::= <scoped_name>
| <literal>
| “(“ <const_exp> “)”

80.*<literal> ::= <integer_literal>
| <string_literal>
| <character_literal>
| <floating_pt_literal>
| <boolean_literal>

81.*<boolean_literal> ::= “**TRUE**”
| “**FALSE**”

82.*<type_dcl> ::= “**typedef**” <type_declarator>
| <struct_type>
| <union_type>
| <enum_type>

83.*<type_declarator> ::= <type_spec> <declarators>

84.*<type_spec> ::= <simple_type_spec>
| <constr_type_spec>

85.*<simple_type_spec> ::= <base_type_spec>
| <template_type_spec>
| <scoped_name>

86.*<base_type_spec> ::= <floating_pt_type>
| <integer_type>
| <char_type>
| <boolean_type>
| <octet_type>
| <any_type>

87.*<floating_pt_type> ::= “**float**”
| “**double**”

88.*<integer_type> ::= <signed_int>
| <unsigned_int>

89.*<signed_int> ::= <signed_long_int>
| <signed_short_int>

90.*<signed_long_int> ::= “**long**”

91.*<signed_short_int> ::= “**short**”

92.*<unsigned_int> ::= <unsigned_long_int>
| <unsigned_short_int>

93.*<unsigned_long_int> ::= “**unsigned**” “**long**”

94.*<unsigned_short_int> ::= “**unsigned**” “**short**”

95.*<char_type> ::= “**char**”

96.*<boolean_type> ::= “**boolean**”

97.*<octet_type> ::= “**octet**”

98.*<any_type> ::= **“any”**

99.*<template_type_spec> ::= <sequence_type>
| <string_type>

100.*<sequence_type> ::= “**sequence**” “<” <simple_type_spec> “,” <positive_int_const> “>”
| “**sequence**” “<” <simple_type_spec> “>”

101.*<string_type> ::= “**string**” “<” <positive_int_const> “>”
| “**string**”

102.*<constr_type_spec> ::= <struct_type>
| <union_type>
| <enum_type>

103.*<struct_type> ::= “**struct**” <identifier> “{“ <member_list> “}”

104.*<member_list> ::= <member>$^{+}$

105.*<member> ::= <type_spec> <declarators> “;”

106.*<union_type> ::= “**union**” <identifier>
“**switch**” “(“ <switch_type_spec> “)”
“{“ <switch_body> “}”

107.*<switch_type_spec> ::= <integer_type>
| <char_type>
| <boolean_type>
| <enum_type>
| <scoped_name>

108.*<switch_body> ::= <case>$^{+}$

109.*<case> ::= <case_label>$^{+}$ <element_spec> “;”

110.*<case_label> ::= “**case**” <const_exp> “:”
| “**default**” “:”

111.*<element_spec> ::= <type_spec> <declarator>

112.*<declarators> ::= <declarator> { “,” <declarator> }*

113.*<declarator> ::= <simple_declarator>
| <complex_declarator>

114.*<simple_declarator> ::= <identifier>

115.*<complex_declarator> ::= <array_declarator>

116.*<array_declarator> ::= <identifier> <fixed_array_size>$^{+}$

117.*<fixed_array_size> ::= “[” <positive_int_const> “]”

118.*<positive_int_const> ::= <const_exp>

119.*<enum_type> ::= “**enum**” <identifier>
“{“ <enumerator> { “,” <enumerator> }* “}”

120.*<enumerator> ::= <identifier>

121.*<except_dcl> ::= “**exception**” <identifier> “{“ <member>* “}”

Appendix B. Further ODL development

This section contains material and issues that are not a part of ODL, but may be in future releases.

B.1 Transactions

An operational interface signature, comprising a set of interrogation and announcement signatures, may be expanded in later versions of ODL to include the following:,

- In the case of interrogation additional information can be provided to design the operation as a transaction initiation or a transaction join.

B.1.1 ODL syntax for Operational Interface Signature

If transaction support is included in ODL, the following syntax may be defined for the signature of operational interfaces.

<oper_dcl> ::= [<trans_attribute>]
<op_type_spec> <identifier>
<parameter_dcls> [<raises_expr>] [<context_expr>]

<trans_attribute> ::= “**transaction**” “**initiation**” | “**transaction**” “**join**”

An alternative approach to enabling transactions in ODL is to pass transaction parameters via an environment object. Currently this is the preferred approach.

B.1.2 Example of Operational Interface Signature

The following example shows the declaration of an operation `delete_line` as a initiation of a transaction.

```
transaction initiation delete_line (in LineId Line);
```

B.2 Contract Template supplement to Object Group Template

In the current version of ODL, a contract is an interface. There is some motivation to provide a contract template that has information additional to an interface template. The specification of a contract template entails specification of the following information:

- Identification of the interface template that forms the basis for the contract template.
- Specification of the release independence policy for the contract template.
- Specification of the security policy governing access to instances of the contract template.

For now, both release independence policy and security policy components are considered as text strings.

B.2.0.1 ODL Syntax for contract declaration

A possible structure for a contract template follows:

<contracts_dcl> ::= <contracts_keywords> <identifier> { “,” <identifier> }* “;”

<contracts_keywords> ::= **“contracts”**

<contract_template> ::= <contract_template_header>
“{“ <contract_template_body> “}”

<contract_template_header> ::= **“contract”** [**“template”**] <identifier>

<contract_template_body> ::= <interf_templ_dcl> <release_dcl> <security_dcl>

<interf_templ_dcl> ::= **“interface”** [**“template”**] <identifier> “;”

<release_dcl> ::= **“release”** [**“policy”**] <string_literal> “;”

<security_dcl> ::= **“security”** [**“policy”**] <string_literal> “;”

B.3 Naming

In a situation where ODL templates and types are developed by disparate organizations, there is likely to be a need to refer to remote templates and types. One way to satisfy this need is to provide a naming scheme for ODL entities that is globally unique.

Support for such a naming scheme may be provided by ODL syntax. For example, if the adopted naming scheme involved a string unique to each organization, (and it identified subparts of that organization) and the globally unique name of an ODL entity was a concatenation of those strings, and the scoped name of the entity’s identifier, then a keyword can be added to ODL to identify the organization (and subpart) on a per file basis.

```
nameHeader <organization_identifier>/<substruct identifier(s)>

group G1{
   ...
   components
      O1, O2;
   ...
   contracts
      I2;

}; // end of G1
```

So the global name of the group template of G1 is:

```
<organization_identifier>/<substruct identifier(s)>/G1
```

The above solution is presented as an example only. It is intended to highlight an area of future development of ODL.

B.4 Multiplicity of Interfaces on Objects, and Objects in Groups

Currently there is no means, in ODL, to specify the initial and maximum number of:

- interfaces on an object, or
- objects in a group, or
- contracts on a group

(other than by making a comment). These issues are considered part of an engineering/ configuration specification, additional to the ODL specification of a system. Hence, this issue is considered outside the scope of this document. However, the production of such an engineering language should be considered an important component of future work.

B.5 Synchronization of Flows

Currently there is no means of indicating whether, and how, flows in a stream interface are synchronized. For example, it would be expected that picture and sound flows in a video phone service have “lip synch”. A cursory investigation of this issue suggests that the problem of specifying synchronization relies on the definition of synchronization. However, there appear to be a number of useful definitions of synchronization, as well as specification of the tolerance of variation that constitutes synchronization. Further investigation is required. before a stable syntax and semantics is identified for ODL. A simple minded means of adding synchronization syntax to ODL is to add a keyword which supports a list of flow identifiers (to indicate which flows are to be in synchronization) and synchronization type identifier. For example, the stream_flow_defn could be changed to the following

122.<stream_sig_defns> ::= { <stream_flow_defn> “;” }$^{+}$ |
[{**“synch”** <synch_type> <flow_identifier_list> ”;” }$^{+}$]

An example of how this could be expressed as ODL is shown below.

```
interface gameServer {
    ...
    sink JoystickFlowType joystick1 with JoystickQos joystick1Qos;
    sink JoystickFlowType joystick2 with JoystickQos joystick2Qos;
    source VideoFlowType picture with VideoQos pictureQos;
    source AudioFlowType sound with AudioQos soundQos;

    synch lipSynch picture sound;
};
```

B.6 Security

Security in TINA systems is an important issue. A security architecture for TINA systems is documented in “Security Architecture” [25], the annex of which includes some suggested additions to ODL to support the proposed architecture. This work needs to be examined in detail and integrated with the current ODL specifications.

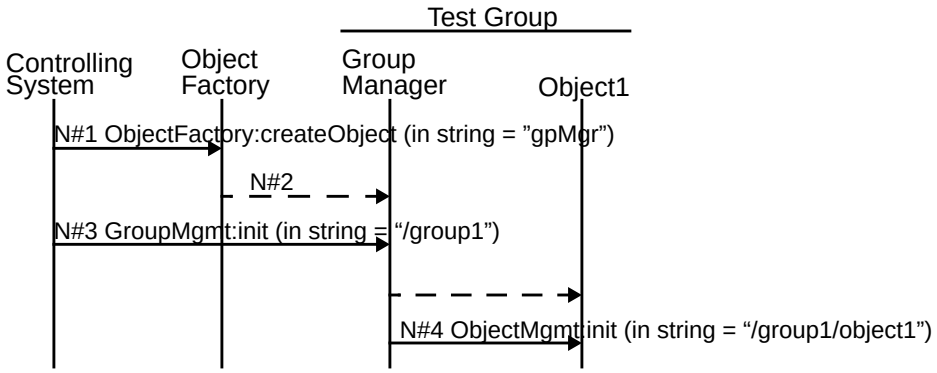
B.7 Instance Interaction Documentation/Diagrams.

ODL is a language is a language describing types. At times there is also a need to document instances of those types, or more specifically, the interactions of instances of ODL types. A notation for documenting such interactions is beyond the scope of this document. However, there is considerable existing literature which can be used as a basis for such documentation, for example, the interaction diagrams of “Object Oriented Software Engineering” [35] and the dynamic model in “Object Oriented Modelling and Design” [36].

Here we also propose the following notation for documenting entity interactions:

B.7.1 Proposed Instance Interaction Documentation Conventions

Following shows an example use of a proposed convention to document a scenario/object-interaction. In this example it involves creation of an object group instance.



N#1 This assumes that the “brand X” object factory service (or similar) is used. The object factory server is deployed at DPE deployment time. The createObject method is given the name of the required object type, and *returns an interface reference to that object’s initial interface*.

N#2 The Group Manager object is created by instantiating a new process (= instantiating a capsule).

N#3 At the top level the Controlling System is responsible for naming the Group manger Instance.

N#4 The Group Manager is responsible for naming the object Instance in the absence of a specific instruction otherwise.

This example uses the following conventions.

Text:

Text above horizontal lines on the diagrams represents object messages, and has the general form (written on one line):

[N#<integer>] [<interfaceName>:]<operationName>([{in, out} <type1>[=value1] ...])

where,

<someTextlabel> = substitute with string

[...] = content between brackets is optional

{comma separated list} = insert one of these options

... = repeat element as needed

The [N#<integer>], for example “N#1” is a reference note marker used to annotate the diagram with text. The integer uniquely identified each note on a per diagram basis.

Many parts of this text have been deliberately made optional to support an iterative development of the diagrams (for example in the absence of ODL). For example, initially the object names and operation names may be identified, then later interface names known, then later still operation parameters and values known, then later still aggregation of objects.

Horizontal Lines:

solid horizontal arrows: These represent RPC invocation from object to object. The return event is not explicitly shown.

dashed horizontal arrows: These represent creation of the object pointed to by the arrow.

solid lines above objects: These denote aggregation of the objects underneath the line.

Others:

Vertical lines represent object instances.

Time is strictly represented by going down the page

B.8 Operations/Flows Common to Multiple Interfaces

In ODL, operations and flows are only visible within the scope of an interface. There may be a need to refer to operations and flows from outside an interface. Such a facility may be used to define an operation that is intended to have the same signature and semantics in multiple interfaces. Currently, two identical operation signatures in different interfaces may be associated with completely different sematics. There are a number of possible ODL modifications which can be introduced to solve this problem. However, currently a work around is proposed, as follows.

If an operation, say op1(), is to have identical semantics in multiple interfaces, say I2 and I3, inheritance can be used to as follows to cater for this:

```
interface I1{
    ...
    void op1 ();
}; // end I1

interface I2:I1{
    ...
    // other operations
}; // end I1

interface I3:I1{
    ...
    // other operations
}; // end I1
```

The disadvantage is that inheritance is used as a syntactic convenience rather than as a reflection of the conceptual domain underlying the system, and there is overhead in introducing the new class. However, these are minimal if the need for common operation semantics in interfaces is not frequent. If the need for such sematics is frequent, modification to ODL should be considered.

Appendix C. Comparison: ODL and OMG-IDL

C.1 ODL Objective vs. OMG-IDL Objective

The objectives of ODL are listed in Section 2.2:

a. Language for application specification (at development time)
b. Language for application re-use (at development time)
c. Language supporting application execution and interaction (at run-time)

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

C.2 TINA Object Model vs. OMG Object Model

The object model of TINA which is supported by ODL (i.e., TINA Computational Modelling Concepts) extends the OMG Object Model in the following ways:

- At the object level, ODL offers support for the definition of:
  - object behaviour.
  - objects with multiple interfaces.
  - object groups.
- At the interface level, TINA-ODL offers support for the definition of:
  - stream interfaces
  - interface behaviour
  - operation and flow QoS requirements
- At the operation and flow level, TINA-ODL offers support for the definition of:
  - stream (flow) signatures

C.3 ODL Syntax vs. OMG-IDL Syntax

ODL in its current version is a superset of OMG-IDL. It implies that as suggested in Section 5.1, it is possible to use OMG-IDL specifications as part of ODL specifications (as operational interface declaration). Consequently, the syntax defined in ODL for operational interface declaration encompasses and supports all the rules defined for OMG-IDL [37].

C.3.1 General Syntax

For an ODL specification, broadly:

- the structures added to OMG-IDL are the object declarations (<object_dcl>), and the object group declarations (<group_dcl>).
- the structures shared with OMG-IDL are the supporting definition (<supporting_def>), and the module declaration (<module>).

- the structure modified from OMG-IDL is the interface declaration (<interface_dcl>).

Below is an extract from the ODL BNF (from Section A) showing the addition to the OMG-IDL syntax. Syntactic rules added to OMG-IDL are in bold, syntactic rules shared between OMG-IDL and TINA-ODL are in italic.

1. <odl_spec> ::= <definition>*
2. <definition> ::= *<module>* ";"
   | **<group_dcl>** ";"
   | **<object_dcl>** ";"
   | <interface_dcl> ";"
   | *<supporting_def>* ";"

C.3.2 Interface Syntax

In the syntax for interface declaration:

- the structures added to OMG-IDL are the (optional) declaration of interface behavior (<interf_behavior_spec>), interface usage specification (<interf_usage_spec>), interface quality of service specification (<interf_QoS_attribute_spec>), trading attributes specification (<trading_attributes_spec>), and the stream (flow) signature declaration (<stream_sig_defns>).
- the structures shared with OMG-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 OMG-IDL is the operation signature declaration (<operation_sig_defns>).

C.3.3 Operation Syntax

In the syntax for operation declaration:

- the structure added to OMG-IDL is the (optional) declaration of QoS constraints on operations (**"with"** <QoS_attributes>).
- the structures shared with OMG-IDL are the announcement declaration (<announcement>), and the interrogation declaration (<interrogation>).

Note that all the additions to the OMG-IDL have been made optional. Below is an extract from the ODL BNF:

<oper_sig_defns> ::= { <oper_sig_defn> ";" }$^{+}$

<oper_sig_defn> ::= *<announcement>*
| <*interrogation*> [**"with" <QoS_attributes>**]

References

TINA-C documents

[1] *Choosing a Computational Specification Notation*, EN TINA-C 1993: EN_A2.FD.002_1.6_93, authors: F.Dupuy, P.Graubmann, N.Natarajan, N.Singer.

[2] *A computational model design process*, EN TINA-C 1993: EN_G.FD.006_1.8_93, authors: F.Dupuy, N.Natarajan.

[3] *Mapping of TINA-C ODL to OMG-IDL*, TINA-C 1994, authors: E.Kelly, G.Wheeler, K.Kanasugi, F.Dupuy.

[4] *Information Modelling Concepts*, Document No. TB_.EAC.001_3.0_94, TINA-C, December 1994.

[5] *Computational Modelling Concepts*, Document No. TB_NAT.002_3.1_94, TINA-C, December 1994.

[6] *Computational Modelling Concepts (1993)*, Document No. TB_A2.NAT.002_3.0_93, TINA-C, December 1993.

[7] *Engineering Modelling Concepts*, Document No. TB_A3.NS.005_1.0_93, TINA-C, December 1993.

[8] *DPE Phase 0.1 Specification* (Section 3: TINA-C IDL), Document No. TB_AD.NW.001_1.0_93, TINA-C, December 1993.

[9] *Engineering Modelling Concepts (DPE Architecture)*, TINA-C 1994, authors: P. Graubmann, W.Hwang, M.Kudela, K.MacKinnon, N.Mercouroff, N.Watanabe.

[10] *What to do with ODL?*, EN TINA-C 1994: EN_G.NM.009_1.08_94, authors: N.Mercouroff, J.Pavon, F.Ruano, path: /u/tinac/94p2/viewable/EN/odl-proposal.ps.

[11] *ODL Inheritance Rules*, EN TINA-C 1995: EN_G.NM.001_1.00_95, authors: N.mercouroff, F.Ruano, path: /u/tinac/94p2/viewable/EN/odl-inheritance.ps

[12] *ODL Syntax for Streams -- a description of the options*, EN TINA-C 1995: EN_BK.001_1.0_1995, author: B.Kitson, path: /u/tinac/95/dpe/viewable/StreamSyntax.ps

[13] *The Factory computational object*, EN TINA-C 1995: EN_G.JP.017_1.01_95, author: J.Pavón, path: /u/tinac/94p2/ccmcomp/notes/factory.ps

[14] *Quality of Service Framework*, Document No. EN_EX.PFM.001_1.0_94, TINA-C, December 1994.

[15] *TINA DPE Service Specifications (Technology Mapping)*, Document No. TR_HW.001_1.0_94, TINA-C, December 1994.

[16] *Packaging*, Document No. TP_HC.010_3.11_94, TINA-C, December 1994.

[17] *TINA-C Service Development Methodology*, Document No. TP_DKB_010_0.1_94,

TINA-C, December 1994.

[18] *Connection Management Specifications*, Document No. TP_NAD.001_1.2_95, TINA-C, March 1995.

[19] *Service Component Specifications*, Document No. TB_HK.002_1.0_94, TINA-C, March 1995.

[20] *Request for Solutions: Naming*, Document No. EN_X.MRS.003_1.0_94, May 1994

[21] *Request for Solutions: Federation*, Document No. EN_G.PG.002_1.0_94, May 1994

[22] *Request for Solutions: Security*, Document No. EN_G.HC.001_5.0_94, May 1994

[23] *Object Grouping and Configuration Management*, Document No. EN_TH.003_1.1_95, August 1995.

[24] *TINA Distributed Processing Environment (TINA DPE)*, TINA-C document No. TR_PL.001_1.3_95, December 1995.

[25] *Security Architecture*, Document No. TR_EGL.004_2.0_1996, March 1996.

[26] *Quality of Service Framework*, Document No. TR_MRK.001_1.0_94, November 1994.

[27] *Computational Modelling Concepts*, Document No. TB_NAT.002_3.2_96, TINA-C, May 1996.

TINA-C Auxiliary Projects

[28] *Programming tools for the PLATyPus experiment*, PLATyPus Project document, version 1.0, 16 March 1994.

[29] *PLATyTools and ODL*, B. Kitson, Proceedings of the TINA'95 Conference, Melbourne, February 1995

Other Related Documents

[30] *Basic Reference Model of Open Distributed Processing, 'Part3: Architecture'*, ITU-T Rec. X.903 | ISO/IEC 10746-3, February 1995.

[31] *Specification and realization of Stream interfaces for the TINA-DPE*, A.T. van Halteren, P. Leydekkers, H.B. Korte, Proceedings of TINA'95 Conference, Melbourne, February 1995.

[32] *Common Object Services Specification*, Volume 1, Revision 1.0, OMG Document Number 94-1-1, OMG, March 1994.

[33] *A Computational and Engineering View on Open Distributed Real-time Multimedia Exchange*, P.Leydekkers, V. Gay, L.J.N. Franken, Proceedings of NOSSDAV'95 (Springer-Verlag), Boston, April, 1995.

[34] *Object Oriented Software Construction*, B. Meyer, Prentice Hall int. series in computer science, Pretice Hall, Hemel Hempstead, UK, 1988.

[35] *Object Oriented Software Engineering*, Jacobson, I., christerson, M., Josson, P. & Overgaard, G., Addison-Wesley, Wockingham, England, 1995.

[36] *Object Oriented Modelling and Design*, Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. &Lorensen, W., Prentice Hall, 1991.

[37] *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, OMG, July 1995.

Acronyms

CS	Capability Set
DPE	Distributed Processing Environment
IDL	Interface Definition Language
ODL	Object Definition Language
OMG	Object Management Group
ORB	Object Request Broker
QoS	Quality of Service
TINA	Telecommunications Information Networking Architecture